Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 95
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 95
Espoo 2005 HUT-TCS-A95

AUTOMATA-THEORETIC AND BOUNDED MODEL CHECKING FOR LINEAR TEMPORAL LOGIC

Timo Latvala

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 95
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 95
Espoo 2005 HUT-TCS-A95

AUTOMATA-THEORETIC AND BOUNDED MODEL CHECKING FOR LINEAR TEMPORAL LOGIC

Timo Latvala

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Computer Science and Engineering, for public examination and debate in Auditorium T2 at Helsinki University of Technology (Espoo, Finland) on the 12th of August, 2005, at 12 o'clock noon.

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FI-02015 TKK
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-7787-5
ISBN 951-22-7788-3 (PDF)
ISSN 1457-7615

Multiprint Oy
Helsinki 2005

ABSTRACT: In this work we study methods for model checking the temporal logic LTL. The focus is on the automata-theoretic approach to model checking and bounded model checking.

We begin by examining automata-theoretic methods to model check LTL safety properties. The model checking problem can be reduced to checking whether the language of a finite state automaton on finite words is empty. We describe an efficient algorithm for generating small finite state automata for so called non-pathological safety properties. The presented implementation is the first tool able to decide whether a formula is non-pathological. The experimental results show that treating safety properties can benefit model checking at very little cost. In addition, we find supporting evidence for the view that minimising the automaton representing the property does not always lead to a small product state space. A deterministic property automaton can result in a smaller product state space even though it might have a larger number states.

Next we investigate modular analysis. Modular analysis is a state space reduction method for modular Petri nets. The method can be used to construct a reduced state space called the synchronisation graph. We devise an on-the-fly automata-theoretic method for model checking the behaviour of a modular Petri net from the synchronisation graph. The solution is based on reducing the model checking problem to an instance of verification with testers. We analyse the tester verification problem and present an efficient on-the-fly algorithm, the first complete solution to tester verification problem, based on generalised nested depth-first search.

We have also studied propositional encodings for bounded model checking LTL. A new simple linear sized encoding is developed and experimentally evaluated. The implementation in the NuSMV2 model checker is competitive with previously presented encodings. We show how to generalise the LTL encoding to a more succint logic: LTL with past operators. The generalised encoding compares favourably with previous encodings for LTL with past operators. Links between bounded model checking and the automata-theoretic approach are also explored.

KEYWORDS: Verfication, Model checking, LTL, automata, safety properties, Petri nets, modular analysis, LTS, testers, bounded model checking, PLTL

CONTENTS

List of Publications vi

1 Introduction 1
1.1 Contributions 6

2 Model Checking LTL 8

3 Automata-theoretic Methods 11
3.1 Model Checking Safety Properties 12
3.1.1 Safety Properties 13
3.1.2 Algorithms 14
3.2 Model Checking Liveness Properties of Modular Petri Nets . 18
3.2.1 Petri Nets and Modular Nets 19
3.3 On-the Fly Verification Using Testers 21
3.3.1 Labelled Transition Systems and Testers 22
3.3.2 Algorithms 23

4 Bounded Model Checking 28
4.1 A New Encoding for BMC 30
4.2 LTL with Past 33

5 Conclusions 37
5.1 Topics for Further Research 37

Bibliography 39

A Additions and Corrections to Publications 53

Articles 55

PREFACE

This Thesis is the result of studies, research and a fair amount of coffee drinking at the Laboratory for Theoretical Computer Science of Helsinki University of Technology.

I am indebted to my supervisor Nisse Husberg for introducing me to the laboratory and for urging me on, especially in the beginning. Discussions with my instructor Keijo Heljanko – usually over coffee – have been excellent brainstorming sessions. His sharp insights and ability to explain complex problems have been invaluable. The head of the laboratory Ilkka Niemelä has not only contributed to the Thesis by securing funding to the laboratory. At least as important is the leadership and scientific integrity he has displayed running the laboratory. My sincerest thanks also go to my co-authors Marko Mäkelä, Heikki Tauriainen, Armin Biere, Tommi Junttila and Keijo Heljanko.

In addition to the the Laboratory for Theoretical Computer Science, this work has been funded by Helsinki Graduate School in Computer Science, National Technology Agency of Finland (TEKES), the Nokia Corporation, EKE Electronics, Genera, the Academy of Finland (projects 47754, 53695, 211025) and by personal grants from the Jenny and Antti Wihuri Foundation, Foundation of Technology (Tekniikan edistämissäätiö) and the Nokia Foundation. Their support is gratefully acknowledged.

My parents and my brother have always supported my aspirations and for this I am grateful. Without my love Emma everything would have been much more difficult.

Otaniemi, July 2005

Timo Latvala

LIST OF PUBLICATIONS

This dissertation consists of five publications and this dissertation summary. In publications [**P1**]–[**P3**] we employ the automata-theoretic approach to model checking LTL in different contexts, and in publications [**P4**]–[**P5**] we investigate efficient encodings for bounded model checking the temporal logic LTL and its generalisation PLTL.

[**P1**] Timo Latvala. Efficient model checking of safety properties. In T. Ball and S. Rajamani, editors, *Model Checking Software. 10th International SPIN Workshop, Portland, Oregon, USA*, volume 2648 of *Lecture Notes in Computer Science*, pp. 74–88, Springer, 2003.

[**P2**] Timo Latvala and Marko Mäkelä. LTL model checking for modular Petri nets. In J. Cortadella and W. Reisig, editors, *Application and Theory of Petri Nets 2004. 25th International Conference, ICATPN 2004, Bologna, Italy*, volume 3099 of *Lecture Notes in Computer Science*, pp. 298–311, Springer, 2004.

[**P3**] Timo Latvala and Heikki Tauriainen. Improved on-the-fly verification with testers. *Nordic Journal of Computing*, 11(2):148–164, 2004

[**P4**] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi Junttila. Simple bounded LTL model checking. In A. Hu and A. Martin, editors *Formal Methods in Computer-Aided Design 2004, 5th International Conference FMCAD 2004, Austin, Texas, USA*, volume 3312 of *Lecture Notes in Computer Science*, pp. 186–200, Springer, 2004.

[**P5**] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi Junttila. Simple is Better: Efficient bounded model checking for past LTL. In R. Cousot, editor, *Verification, Model Checking and Abstract Interpretation 2005, 6th International Conference VMCAI 2005, Paris, France*, volume 3385 *Lecture Notes in Computer Science*, pp. 380–395, Springer, 2005.

The current author is the sole author of [**P1**] and has played a major role in all other publications.

Publication [**P2**] was co-authored by Marko Mäkelä. The key ideas and proofs were developed by the current author and the paper was to a large extent written by the current author. The implementation and the experiments were the co-author's work.

Publication [**P3**] was co-authored by Heikki Tauriainen. The original idea and initial versions of the algorithm were developed by the current author. The paper was jointly written, and the correctness proof for the algorithm and certain improvements to the algorithm were contributed by the co-author.

Publication [**P4**] was co-authored by Armin Biere, Keijo Heljanko and Tommi Junttila. The BMC encoding developed was to a large extent joint work with K. Heljanko, starting from initial versions by K. Heljanko. The current author had the main responsibility for writing the paper. The analysis of the original encoding in Section 2 was written by Armin Biere. The implementation and the experiments were contributed by the author.

Publication [**P5**] was co-authored by Armin Biere, Keijo Heljanko, and Tommi Junttila. The BMC encoding developed was to a large extent joint work with K. Heljanko, starting from initial versions by the author. The current author had the main responsibility for writing the paper. Also the implementation and the experiments are work of the current author.

1 INTRODUCTION

We are moving towards a society where computers and software are increasingly managing business and safety critical functions. In effect, we are relying on our computers and software to function correctly more than ever. At the same time, the environment our systems are functioning in is becoming more challenging. There is complexity in the form of concurrency, heterogeneity, changing requirements and numerous other sources.

Of the above mentioned sources of complexity perhaps the most studied is concurrency. Concurrency can be an issue at many levels of a complex system. A modern telecommunication system must resolve concurrency issues between the participating devices of the network. At the same time, the mobile devices themselves are also concurrent systems: a mobile phone may be receiving a text message while the user is adding an item to the calendar. Another common feature of many complex systems is reactivity. A reactive system receives inputs from the environment and continuously reacts to the changing situation. Typical reactive systems include embedded controllers, mobile phones or computer operating systems.

Modelling reactive systems is challenging. A reactive system cannot be adequately described by the traditional model of behaviour where a system receives inputs and performs finite computations to produce output. Even small reactive and concurrent systems can be very complex to analyse: the combinatorial explosion of possible states in the system makes analysing the state space of the system hard. In addition, the inherent non-determinism in these systems contributes to the combinatorial explosion and complicates analysis in many ways. For systems implemented as hardware the sources of complexity are similar. The dominating factor causing state explosion is the large number of internal states combined with the non-determinism due to many free input variables in the system.

Despite the increasing complexity of systems there is an expectation of increased productivity. Thus, understanding and analysing concurrent and reactive systems is a challenging but necessary endeavour. Currently the most widely used verification and validation method is testing. However, the non-determinism inherent in reactive and concurrent systems severely degrades the performance of methods such as testing. Repeating failing executions is very difficult in the presence of non-determinism, and this hinders effective regression testing and debugging. The large state spaces usually associated with concurrent systems and hardware systems also make achieving good coverage of the possible behaviours hard: even measuring coverage is difficult. The overall increase in the complexity of systems and the inadequacy of testing have been suggested as reasons for the faint signs that productivity in the software industry is actually decreasing [75].

An active research direction aiming at increasing the productivity of system designers is the research on *computer aided verification*. In this framework one analyses a mathematical *model* of the system and the environment w.r.t. a formal specification. The implicit assumption is that the mathematical model of the system reflects the important aspects of the real system. The analysis of the properties of the model is, however, only valid for the real system as far as the model reflects the properties of the real system. Computer analysis of the relationship between the system model and the specification, as well as various methods for obtaining the model and the specification can be seen as one definition of computer aided verification.

In 1977 Amir Pnueli [133] suggested that *temporal logic*, a variant of modal logic, could be suitable for specifying properties of reactive and concurrent systems. Pnueli argued that temporal logic provided a natural mathematical framework for reasoning about systems with non-terminating infinite behaviour. The specific variant of temporal logic Pnueli advocated modelled a system as a set of infinite sequences of states (executions). The notion of time in this model is discrete, and in an execution each time point has a unique future. The specification language Pnueli presented is called linear temporal logic (LTL).

Using temporal logic as a specification language for systems quite naturally leads to the idea of model checking. *Model checking* [31, 134, 36] is one of the most actively studied computer aided verification techniques. In model checking a temporal logic specification is checked against a *Kripke* model of the system. Kripke models of reactive systems capture the continuous interaction with the environment by modelling it with infinite computations. However, the interaction with the environment must be explicitly modelled since Kripke models do not have a notion of input or output w.r.t. the environment. The Kripke model represents the discrete state transition behaviour as a directed labelled graph. Computations of the system are infinite paths in the graph. Temporal logic is used to specify the allowed computations of the system. The computations allowed by the temporal logic specification can be mechanically compared with the computations of the Kripke model of the system. This process is called model checking.

Model checking has several attractive features. Once the Kripke model of the system and the temporal logic specifications have been defined (a non-trivial task in itself), the process is fully automatic. If a computation that violates the specification is found in the Kripke model, it can be displayed to the system designer to aid the debugging process [35, 119, 74]. In addition, the model checker can be designed to display a proof if the temporal logic property holds [128, 131, 113].

An alternative approach to model checking for proving temporal logic properties of a system is to use *deductive verification*. In deductive verification both the system and the properties are formalised as logical formulae. A theorem prover is used to show that the system has the desired properties. Deductive verification is often criticised for requiring highly educated experts to complete the proofs: the theorem provers are solving undecidable problems and in many cases need manual intervention to succeed. Model checking can, at least partly, be criticised on the same grounds. Although model checking is in principle automatic, manual abstraction is often required for a successful model checking effort. Deductive verification has been the method of choice for proving properties for some classes of systems such as general parametric systems. However, a recent result shows that model checking combined with finitary abstraction is as powerful as deductive verification [108]. Thus, because it is easier (in the current author's opinion) to find abstractions of systems than to find powerful invariants for automated proofs, model checking has greater potential than deductive verification.

Perhaps the greatest hurdle to ubiquitous industrial use of model checking is the *state explosion problem* (see [164] for a review). Even for simple models of concurrency, e.g. 1-safe Petri nets [48], analysing fundamental properties such as reachability of illegal states, is PSPACE-complete[1] (see e.g. [58]). Concurrent systems usually exhibit a combinatorial explosion in the number of global system states, because events in the system can occur in many possible ways as the participating processes are fairly independent. Research into alleviating the state explosion problem has been and remains one of the most active areas of study. Different approaches to alleviate the state explosion problem include symbolic model checking [24, 126, 14], partial order methods [162, 106, 70, 170], complete finite prefixes [125, 126, 84], compositional methods [38, 76, 121, 103], symmetry reduction [98, 78, 56, 140, 102] and abstraction [43, 34, 45, 71].

Although the state explosion problem is a great challenge, other issues also impede the industrial use of model checking. Constructing an accurate model of the system under inspection is a resource intensive task that can require the largest portion of the resources of a model checking effort [57, 138, 161]. For software systems obstacles faced when modelling a system include how to model advanced features of programming languages such as pointers, dynamic memory allocation and recursion. Another problem is ensuring the accuracy of the model when modelling a large system. Hardware systems mostly lack the complex features of software systems, but the large size of the systems makes isolating a manageable subsystem difficult. Many attempts at solving the modelling problem are based on automating the model construction phase in the model checking effort [91, 40, 9, 95].

Model checking requires that the specification is formulated in a temporal logic. Unfortunately, expressing the desired property accurately can be challenging even for a practitioner. False model checking results due to erroneous formulas are not unusual in model checking efforts [89, 91, 57]. Several approaches have been suggested to alleviate these problems: providing templates of useful formulas [50]; adding syntactic sugar to the logic and extending the language [10, 6]; providing a graphical front-end for LTL [152] and developing a domain specific front-end for the property language [41].

In some cases expressing the required properties is impossible because the logic which is used does not have the required expressive power. LTL and its simple extensions can only specify a proper subset of the ω-regular properties called the star-free ω-regular properties [159]. Lichtenstein et al. [120] have argued that full ω-regularity is useful for facilitating compositional reasoning. Furthermore, there are interesting properties which are not even regular. For instance, expressing pre-condition, post-condition style properties for functions in formalisms where recursion is allowed is not a regular property [5]. Some research has been conducted in model checking non-regular specification formalisms. In [5] a method for model checking recursive state machines for a non-regular logic is developed, while [111] presents model checking of pushdown specifications for regular systems.

[1] Here and in the following we use complexity classes as defined in [130].

The focus of this work has been to develop efficient model checking methods. The main techniques we have investigated are the automata-theoretic approach to model checking and bounded model checking. We have applied the automata-theoretic framework to different domains with special emphasis on the algorithmics for model checking LTL. In bounded model checking we have focused on developing efficient encodings of the LTL model checking problem into propositional logic.

The automata-theoretic approach to model checking [115, 168, 169, 166] presents automata as a uniform approach to specification and verification. The approach exploits the close connection between temporal logic and automata on infinite words. Any LTL property can be expressed by a so called *Büchi* automaton [168, 67]. In this approach automata are used to model both the system under inspection and the specification. Interesting questions, e.g. whether the behaviour of the system obeys the specification, can easily be cast into automata-theoretic terms. In many cases asymptotically optimal algorithms have been obtained using the automata-theoretic approach. Since automata are essentially labelled graphs, many of the key algorithms are actually adapted graph algorithms.

An important class of properties are the so called *safety properties*. Safety properties are properties of the system that have finite counterexamples or, more informally, properties requiring that “nothing bad happens”. A safety property can e.g. express that “if the variable y becomes negative at some point, then the Boolean variable x will have been false before this occurs”. As safety properties include properties such as invariants, they are usually considered the most important and critical subset of properties to verify in a system. Another reason safety properties are interesting is that algorithms for model checking safety properties are simpler and more efficient than algorithms for the general case. The automata-theoretic approach accommodates treating safety properties as a special case. Standard finite automata on finite words can detect counterexamples to LTL safety properties while so called Büchi automata are required by general LTL properties. The relevant algorithms for finite automata are simpler than the corresponding ones for Büchi automata. In some cases the difference can be so radical that model checking safety properties is decidable, while model checking general properties is not: LTL safety properties are decidable for (unbounded) Place/Transition nets, while general (state based) LTL properties are not [58]. Safety properties are also interesting outside the model checking context. In runtime analysis of programs [83] only safety properties can be monitored, since liveness properties can only be refuted by an infinite trace.

Many of the previously mentioned benefits can be obtained by having a compact deterministic finite automaton expressing the LTL safety property. We have investigated the problem of generating a finite automaton from an LTL safety specification in [**P1**].

An attractive feature of the automata-theoretic approach to model checking is its ability to flexibly adapt to different frameworks. A popular way to describe the behaviour of a distributed system is to use Petri nets [136]. In their most basic form, Petri nets do not include structural information such as the system’s partition into processes. One extension of Petri nets that adds process structure to Petri nets is called modular Petri nets [26]. This additional structural information can be exploited by methods that attempt to alleviate the state explosion problem. *Modular analysis* is a method to generate a smaller state space for modular Petri nets such that the reduced state space still contains the relevant information for analysing the behaviour of the system. In publication [**P2**] we present an automata-theoretic LTL model checking method that is compatible with modular analysis.

The model checking method developed in [**P2**] reduces model checking modular Petri nets to a special case of *verification with testers* [163, 81]. Testers are a form of automata suitable for defining illegal behaviours of labelled transition systems (LTSs). LTSs model systems as finite sets of synchronising processes. Each process is modelled by a finite state automaton. Testers are LTSs with additional structural information to enable them to express both liveness and safety properties. The basic algorithmic problems related to verification with testers had not been characterised fully until [**P3**]. In [**P3**] we study the tester verification problem and present and prove correct an efficient algorithm for solving the tester verification problem.

Symbolic model checking [24, 126] is an efficient way of alleviating the state explosion problem. The basic idea is to represent the state space implicitly using symbolic means. In their seminal paper Burch et al. [24] used propositional logic formulae manipulated with (ordered) Binary Decision Diagrams (BDDs). Using this arrangement they succeeded in model checking systems with unprecedently large state spaces. BDDs are a canonical representation for Boolean formulas. They can succinctly represent many Boolean functions. However, the efficiency of the BDD representation for a function is dependent on finding good *variable ordering* for the variables in the Boolean formula. This can be difficult, and there are functions which do not have any succinct BDD representation [22]. The problem manifests itself in model checking when the BDD representing the currently reachable state space (or equivalently the Boolean formula) is computed. Unpredictable blow-ups in memory usage can occur if the current variable order is unsuitable for the current Boolean formula. For this reason, several variable ordering heuristics [137] and partitioning methodologies [23, 129] have been developed for BDDs.

In bounded model checking [14] (BMC) a limited model checking problem is considered: only counterexamples of a fixed length k are sought for. By letting the bound k grow incrementally we can prove that the systems contains no counterexamples of length k or shorter. For a finite state system the method is complete if one lets the bound grow large enough. However, determining exactly how large the bound must be is a hard problem. Bounded model checking uses the same basic idea as symbolic model checking with BDDs: the state space of the system is represented implicitly using Boolean formulas. However, Boolean formulas are not represented using a canonical form. Instead the BMC problem defined above is mapped to the propositional satisfiability problem (SAT). BMC has its roots in similar methods employed for AI planning problems [107]. Given a system M, a temporal logic formula ψ and a bound k, a Boolean formula is constructed which is satisfiable if and only if M has a counterexample of length k to ψ. A propositional satisfiability solver is used to perform the query. Solving the SAT problem from a BDD representation is easy, but the canonical representation of Boolean formulas can unexpectedly blow-up. In BMC, the growth of the size of the BMC formulas can be known in advance, but predicting the running times of the SAT solver is difficult. BMC is particularly good at finding short counterexamples, and there are several papers describing successful industrial applications of BMC [15, 39, 17]. The size and efficiency of the encoding into propositional logic affects the performance of BMC. In [**P4, P5**] we have investigated efficient encodings for LTL and its generalisation PLTL.

There are also other ways to implement symbolic model checking without BDDs. Many of these are attempts to remedy the problems encountered when using BDDs. Verification using Boolean expression diagrams [172] (BEDs) aims at combining the advantage of BDDs and compact propositional logic expressions by combining them in a hybrid data structure. The use of BDDs allows easily checking whether the complete state space has been explored, and the compact propositional representation alleviates the space requirements of the BDDs. Another approach is to strictly use an efficient propositional representation as in [2]. Symbolic reachability analysis is achieved by using quantifier elimination, which can, however, be time consuming and cause a blow-up in the propositional expression. Quantifier elimination can be avoided if a SAT solver that can return all possible solutions to the propositional formula is used. This approach is used in [77]. However, the representation of all solutions as a Boolean formula can be exponential in the number of propositional variables.

1.1 CONTRIBUTIONS

In this dissertation we study methods and algorithms that aim at facilitating more efficient verification, especially model checking. Publications [**P1**]–[**P3**] describe ways to improve model checking and verification using the automata-theoretic approach. In publications [**P4**]–[**P5**] we discuss novel methods for bounded model checking.

The main contributions of each publication are the following:

[**P1**] An algorithm based on [112] for translating LTL safety formulas to finite automata is presented. Experimental results indicate that the tool is competitive compared to tools translating LTL to Büchi automata, and that treating safety as a special case improves performance. Additionally, the implemented translation tool *scheck* can identify *pathological* safety formulas. Pathological safety formulas are safety formulas without (comparatively) succinct bad prefixes and therefore are inefficient to model check using the same techniques as other safety formulas.

[**P2**] Modular analysis is a way of alleviating the state explosion problem for modular Petri nets. An automata-theoretic method for model checking the temporal logic LTL-X compatible with modular analysis is presented. The method retains the ability of modular analysis to exploit invisible transitions to reduce the state space. Experimental results show the same mixed results as for modular analysis in general.

[**P3**] Testers are an alternative formalism to LTL and Büchi automata for specifying properties of a reactive system. They have a more refined view of visibility than standard Büchi automata synchronisation, and can therefore allow a more fine grained treatment of concurrency to alleviate the state explosion problem. The problem of on-the-fly verification with testers is analysed and solved. An efficient on-the-fly algorithm is presented an proved correct.

[**P4**] A succinct SAT encoding of the bounded LTL model checking is presented. The size of the encoding is linear in the bound and the size of the formula. The encoding has been implemented on top of the NuSMV2 model checker [28]. Experiments show that the encoding scales better than previous encodings both in the length of bound and the size of the formula.

[**P5**] Based on the LTL encoding in [**P4**], a SAT encoding of the bounded PLTL model checking is presented. The size of the encoding is linear in the bound and quadratic in the size of the formula. However, if the nesting depth of past operators is fixed the encoding is also linear in the size of the formula. The experimental results confirm that the encoding produces smaller SAT instances and that SAT solvers are able to return results faster than previous implementations.

The Structure of the Dissertation The dissertation consists of five publications and this dissertation summary, and has the following structure.

In Section 2 we introduce the most important concepts in model checking LTL, the main theme of this dissertation. The automata-theoretic approach to model checking, with special emphasis on results related to publications [**P1**]–[**P3**] are presented in Section 3. In Section 4 we discuss bounded model checking and describe the results of publications [**P4**]–[**P5**]. Conclusions and future work are discussed in Section 5.

2 MODEL CHECKING LTL

The basic problem this work examines is model checking the temporal logic LTL. In this section we introduce the notation used throughout the text and define the central problems.

We use Kripke structures in their most basic form as the model of computation. The model is appropriate for reactive computation where the relative order of events matters but the absolute time difference between events does not. We restrict our attention to finite state systems. Although this is a fairly limited model it can with appropriate abstractions capture a large class of systems.

We define a Kripke structure in the following way.

Definition 1 A Kripke *structure is tuple* $M = \langle S, \delta, s_0, l \rangle$, *where*

- S *is a finite set of states,*
- $\delta \subseteq S \times S$ *is the transition relation such that for all* $s \in S$ *there exists* $s' \in S$ *with* $(s, s') \in \delta$,
- $s_0 \in S$ *is the initial state and*
- $l : S \rightarrow 2^{AP}$ *is a labelling function, where* AP *is a set of atomic propositions.*

An execution *is an infinite sequence of states* $\sigma = s_0 s_1 s_2 \ldots$ *such that* s_0 *is the initial state and* $(s_i, s_{i+1}) \in \delta$ *for all* $i \geq 0$.

A Kripke structure can be seen as defining a language of infinite words over 2^{AP}, a fact which is exploited by the automata theoretic approach to model checking. The language of a Kripke structure M is denoted $\mathcal{L}(M)$ and defined by $\mathcal{L}(M) = \{l(\sigma) \mid \sigma \text{ is an excution of } M\}$, where we have generalised $l(s)$ to sequences in the natural way. Executions σ projected with the labelling function $l(\sigma)$ are referred to as *computations*. Although we only allow non-terminating executions we can model deadlocking executions by adding a selfloop to a deadlocking state.

We use the temporal logic LTL to specify legal computations of a Kripke structure. An LTL formula φ is defined over a set of atomic propositions AP. The syntax of LTL is given by the following inductive definition:

1. $\psi \in AP$ is an LTL formula.
2. If ψ_1 and ψ_2 are LTL formulae then so are $\neg\psi_1$, $\mathbf{X}\,\psi_1$, $\psi_1\,\mathbf{U}\,\psi_2$, $\psi_1\,\mathbf{R}\,\psi_2$, $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$.

The temporal operators are the next-time operator $\mathbf{X}$, the until-operator $\mathbf{U}$, and its dual the release-operator $\mathbf{R}$. Given an LTL formula ψ, the set of unique subformulas of ψ is denoted by $cl(\psi)$ and the shorthand $|\psi|$ denotes the cardinality of $cl(\psi)$.

The semantics of LTL is is defined over computations $\pi = \sigma_0\sigma_1\sigma_2\ldots$ with $\sigma_i \in 2^{AP}$. We say that π is a model of ψ at position i, denoted $\pi^i \models \psi$, when the inductively defined relation '$\models$' holds:

$$\begin{array}{lll}
\pi^i \models \psi & \Leftrightarrow & \psi \in \sigma_i \text{ for } \psi \in AP. \\
\pi^i \models \neg\psi & \Leftrightarrow & \pi^i \not\models \psi. \\
\pi^i \models \psi_1 \vee \psi_2 & \Leftrightarrow & \pi^i \models \psi_1 \text{ or } \pi^i \models \psi_2. \\
\pi^i \models \psi_1 \wedge \psi_2 & \Leftrightarrow & \pi^i \models \psi_1 \text{ and } \pi^i \models \psi_2. \\
\pi^i \models \mathbf{X}\,\psi & \Leftrightarrow & \pi^{i+1} \models \psi. \\
\pi^i \models \psi_1\,\mathbf{U}\,\psi_2 & \Leftrightarrow & \exists n \geq i:\ \pi^n \models \psi_2 \text{ and } \pi^j \models \psi_1 \text{ for all } i \leq j < n. \\
\pi^i \models \psi_1\,\mathbf{R}\,\psi_2 & \Leftrightarrow & \forall n \geq i:\ \pi^n \models \psi_2 \text{ or } \pi^j \models \psi_1 \text{ for some } i \leq j < n.
\end{array}$$

When $\pi^0 \models \psi$ we usually simply write $\pi \models \psi$. Commonly used abbreviations are the standard Boolean shorthands $\top \equiv p \vee \neg p$ for some $p \in AP$, $\bot \equiv \neg\top$, $p \Rightarrow q \equiv \neg p \vee q$, $p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$, and the derived temporal operators $\mathbf{F}\,\psi \equiv \top\,\mathbf{U}\,\psi$ ('finally'), $\mathbf{G}\,\psi \equiv \neg\mathbf{F}\,\neg\psi$ ('globally'). We use LTL-X to denote the subset of LTL without the next-operator $\mathbf{X}$.

In some cases it is desirable to rewrite a formula to *positive normal form*, where negation only occurs in front of atomic propositions. This can be accomplished using the dualities $\neg\,(\psi_1\,\mathbf{U}\,\psi_2) \equiv \neg\psi_1\,\mathbf{R}\,\neg\psi_2$, $\neg\,(\psi_1\,\mathbf{R}\,\psi_2) \equiv \neg\psi_1\,\mathbf{U}\,\neg\psi_2$, $\neg\mathbf{X}\,\psi \equiv \mathbf{X}\,\neg\psi$ and de Morgan's laws for propositional logic.

The basic problem of satisfiability for LTL, i.e. does an LTL formula ψ have any model π such that $\pi \models \psi$ is PSPACE-complete in the size of the formula [151]. Determining if an LTL formula accepts all models, the validity problem, can be solved by negating the given formula and solving the satisfiability problem. Validity is also a PSPACE-complete problem [151].

There are two interesting model checking problems w.r.t. Kripke structures. The so called existential model checking problem asks, given a Kripke structure M and an LTL formula ψ, does M have a computation π such that $\pi \models \psi$. The dual of the existential problem is the universal problem that asks if all computations π of M are models of ψ (denoted $M \models \psi$). The problems are dual in the sense that $M \not\models \psi$ iff $\pi \models \neg\psi$ for some computation π of M. Solving the model checking problem for Kripke structures is PSPACE-complete in the size of the formula [151, 168]. The complexity of the problem is only linear in the size of the Kripke structure $|M|$, but $|M|$ can be very large due to the state explosion problem. The precise complexity of model checking a single ultimately periodic computation, i.e. deciding whether $\pi \models \psi$, where $\pi = u(v)^\omega$ and $u, v \in (2^{AP})^*$, is currently unknown [118]. It is an interesting problem because LTL model checking can be decided by only studying ultimately periodic computations [168]. The problem is in P and at least NC$_1$-hard [118], but efforts to prove the problem NL-hard have so far failed. For the at least exponentially more succinct logic NLTL (a linear temporal logic with forgettable past) the problem is P-complete [118].

An LTL formula ψ defines a language of infinite words over 2^{AP} given by $\mathcal{L}(\psi) = \{\pi \in (2^{AP})^\omega \mid \pi \models \psi\}$. Consequently, when LTL is used as a specification language, the language of the formula can be seen as defining the legal computations of the Kripke structure. In this view a Kripke structure is a model of an LTL formula ψ iff $\mathcal{L}(M) \subseteq \mathcal{L}(\psi)$. To solve the satisfiability and validity problems for LTL, we must decide whether $\mathcal{L}(\psi) = \emptyset$ or $\mathcal{L}(\psi) = (2^{AP})^\omega$ respectively.

Two important logics that LTL is often compared with are the branching-time logics CTL and CTL* (see e.g. [36]). CTL* is a strict superset of LTL, where a formula can contain arbitrarily nested path quantifiers. The path quantifiers express existential or universal choice over paths and can be used to express properties of the form "for all paths in the system, there exists a path to the initial state from any state of the path". Model checking CTL* is PSPACE-complete [54]. CTL is a syntactic restriction of CTL* that is important because it has polynomial time model checking algorithms w.r.t. the size of the formula [32].

Although model checking CTL is easier than model checking LTL and CTL* is more expressive than LTL while retaining the same model checking complexity, this work focuses on LTL and its extensions. Vardi [167] has argued that the linear-time framework is superior to the branching-time framework for several reasons. We repeat some of the reasons here. In practice CTL has proven to be difficult to use. Engineers find the branching nature of time unintuitive. Another important factor is compositional reasoning. CTL is neither expressive enough for compositional reasoning, nor does have it have a complexity theoretical advantage in this domain. All interesting questions in compositional model checking are at least PSPACE-complete for CTL w.r.t. the size of the formula (the same complexity as LTL). There are also issues that are specific to the research conducted in this work. For branching-time properties it is an open question whether there is any advantage in treating safety properties as a special case [112]. Bounded model checking fits naturally in the linear-time framework while it is not compatible with the general branching-time framework. The work on BMC for branching time has exclusively focused on the universal fragments of branching time temporal logics [132, 174].

3 AUTOMATA-THEORETIC METHODS

The automata-theoretic approach to model checking [115, 168, 166] exploits the close connection between LTL and automata on infinite words. By viewing both the system and the specification as automata, the approach provides a uniform way of deciding essentially all problems related to model checking LTL.

The connection between LTL and automata on infinite words is usually established through languages. An LTL formula defines a language over infinite strings that can be accepted by an automaton on infinite words. The most basic type of automaton on infinite words is called a Büchi automaton.

Let $w \in \Sigma^\omega$ be an infinite word over the alphabet Σ. A *Büchi* automaton is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, F \rangle$, where Σ is the *alphabet*, Q is a finite set of *states*, $\delta : (Q \times \Sigma) \to 2^Q$ is the *transition function*, $Q_0 \subseteq Q$ the set of *initial states*, and $F \subseteq Q$ is a set of *accepting states*. A *run* of the automaton $\mathcal{A}$ on a word $w = \sigma_0\sigma_1\sigma_2 \ldots \in \Sigma^\omega$ is a mapping $\rho : \mathbb{N} \to Q$ such that $\rho(0) \in Q_0$ and $\rho(i+1) \in \delta(\rho(i), \sigma_i)$ for all $i \geq 0$. Let $\mathit{inf}(\rho)$ to denote the set of states occurring infinitely often in the run. A word w is accepted if there is a run ρ on w such that $\mathit{inf}(\rho) \cap F \neq \emptyset$.

A *generalised* Büchi automaton generalises the acceptance condition of a standard Büchi automaton. The single set of accepting states F is replaced by a family of sets $\mathcal{F} = \{F_1, F_2, \ldots, F_k\}$. A word w is accepted if there is a run ρ on w such that $\bigwedge_{i=1}^{k} \mathit{inf}(\rho) \cap F_i \neq \emptyset$.

A number of papers have considered the problem of efficiently generating a Büchi automaton $\mathcal{A}$ that accepts exactly the same language as a given LTL formula (see e.g. [67, 44, 64, 156]). The worst case complexity for the size of the automaton is exponential in the size of the formula, which is not surprising since the LTL satisfiability problem can be solved in linear time w.r.t. the size of Büchi automaton representing the LTL formula. Recall that for an LTL formula ψ and a system M, the property ψ holds in the system iff $\mathcal{L}(M) \subseteq \mathcal{L}(\psi)$. This is equivalent to $\mathcal{L}(M) \cap \mathcal{L}(\neg\psi) = \emptyset$. From this it is straight-forward to derive an automata-theoretic approach to solve the model checking problem. We start by constructing the Büchi automaton $\mathcal{A}_{\neg\psi}$ for the negation of the property ψ. Next, the product $M \times \mathcal{A}_{\neg\psi}$ is computed which accepts the intersection of the languages $\mathcal{L}(\mathcal{A}_{\neg\psi}) \cap \mathcal{L}(M)$ of the two automata M and $\mathcal{A}_{\neg\psi}$. Clearly, $M \models \psi$ if and only if $\mathcal{L}(M \times \mathcal{A}_{\neg\psi}) = \emptyset$. This can be checked by fairly simple graph algorithms, since the language of a Büchi automaton is empty iff there is no reachable accepting state $q \in F$ such that q can be reached from itself by a non-empty path. We can thus reduce the LTL model checking problem to checking if the language of the product automaton is empty. This is referred to as an emptiness check of an automaton on infinite words.

In the previously presented approach it is important that we can directly construct the automaton for the negation of the property. Complementing a Büchi automaton is complicated and has an exponential ($2^{O(n \log n)}$) worst case lower bound [139]. If the automaton for the LTL formula is deterministic, complementing the Büchi automaton is a linear-time procedure that only doubles the size of the automaton [114]. However, deterministic Büchi automata cannot express all LTL properties (see e.g. [166]).

There are several possible ways of determining whether $\mathcal{L}(M \times \mathcal{A}_{\neg\psi}) = \emptyset$. If we are working in a explicit state context the most common solutions either use some variant of the nested depth-first algorithm [42, 90] or an algorithm based on Tarjan's algorithm to compute the maximal strongly connected components (SCCs) for a directed graph [155]. An SCC is a maximal subset of states $C \subseteq Q$ of the automaton such that for all $q_i, q_j \in C$ there is a path from q_i to q_j and vice versa. Both approaches solve the emptiness problem in linear time w.r.t. the size of the automaton. Section 3.3 discusses explicit state algorithms for checking emptiness in detail.

3.1 MODEL CHECKING SAFETY PROPERTIES

A very interesting subclass of all properties specifiable with LTL (or more generally all ω-regular properties) are those for which *all* counterexamples are finite. LTL is defined over infinite words but a finite counterexample can be understood as saying that any continuation of a finite counterexample is also a counterexample. For this subclass of properties, it is possible to detect violations of the property by analysing only finite computations. Thus, these properties in one sense require that "nothing bad happens", because once a system leaves the set of safe states the property is irrevocably violated. The properties are therefore referred to as *safety properties*. Properties for which *all* counterexamples are infinite are called liveness properties. Any property can be expressed as a conjunction of a safety and a liveness property [3]. More details on the classification can be found in [4].

Since safety properties have finite counterexamples, we can for each LTL safety property construct a standard automaton on finite words (FSA) that accepts all violating computations for the given property. This automaton can be doubly exponential in the size of the formula in the worst case [112]. However, if we restrict ourselves to safety properties with so called informative counterexamples (also called non-pathological formulas), only a worst case singly exponential automaton is required. This is interesting because the algorithmics for finite automata is simpler than for Büchi automata. The language of a standard finite automaton is non-empty if some final state can be reached. Model checking safety formulas can therefore be reduced to reachability. Compared with general LTL properties where model checking is reduced through Büchi automata to repeated reachability of a set of accepting states, model checking safety properties is much simpler.

Model checking safety properties is an interesting special case of the general model checking problem. Although model checking finite state systems remains PSPACE-complete w.r.t. the given LTL safety formula [150], the simpler algorithms required for model checking safety properties can in some cases reduce an undecidable problem to a decidable one. One concrete example is model checking state-based LTL for (unbounded) Petri nets [58]. Another example, where the difference is not quite as radical, is BDD-based model checking. Solving the problem for the safety case can in practice be considerably easier than the general case [61], although the difference in the number of image computations is linear vs. $n \log n$ [18] or quadratic [53], depending on the used approach. Also for finite complete prefixes safety is much easier than the general case [84].

There are three basic problems to be solved when applying the automata-theoretic approach to model checking safety properties:

- Deciding if the given LTL specification is actually a safety property.
- Deciding if the given LTL specification is pathologically safe.
- Constructing an FSA accepting all computations violating the safety specification.

The two first problems are PSPACE-complete in the size of the formula [150, 112].

3.1.1 Safety Properties

LTL safety properties can be formally defined using the language characterisation of properties (see e.g. [112]). Let $L \subseteq \Sigma^\omega$ be a language on infinite words over an alphabet Σ. We say that a finite word $x \in \Sigma^*$ is a *bad prefix* for a language L, if for every $y \in \Sigma^\omega$ we have that $x \cdot y \notin L$. Given a language L, if all $w \in \Sigma^\omega \setminus L$ have a bad prefix we call L a *safety language*. An LTL formula is a safety property if it defines a safety language.

Safety properties with only informative counterexamples are of special interest, since a singly exponential FSA can capture all counterexamples (bad prefixes). An informative counterexample for an LTL formula can be seen as explaining the failure of each subformula for the given formula. All subformulas in the original formula have a reason for not holding. Let ψ be an LTL formula in positive normal form and π a finite computation $\pi = \sigma_0\sigma_1 \ldots \sigma_n$. The computation π is *informative* for ψ iff there exists a mapping $L : \{0, \ldots, n+1\} \rightarrow 2^{cl(\neg\psi)}$ such that the following conditions hold:

- $\neg\psi \in L(0)$,
- $L(n+1)$ is empty, and
- for all $0 \leq i \leq n$ and $\varphi \in L(i)$, the following hold.
  - If φ is an atomic proposition, then $\varphi \in \sigma_i$.
  - If $\varphi = \varphi_1 \vee \varphi_2$, then $\varphi_1 \in L(i)$ or $\varphi_2 \in L(i)$.
  - If $\varphi = \varphi_1 \wedge \varphi_2$, then $\varphi_1 \in L(i)$ and $\varphi_2 \in L(i)$.
  - If $\varphi = \mathbf{X}\,\varphi_1$, then $\varphi_1 \in L(i+1)$.
  - If $\varphi = \varphi_1 \,\mathbf{U}\, \varphi_2$, then (i) $\varphi_2 \in L(i)$ or (ii) ($\varphi_1 \in L(i)$ and $\varphi_1 \,\mathbf{U}\, \varphi_2 \in L(i+1)$).
  - If $\varphi = \varphi_1 \,\mathbf{R}\, \varphi_2$, then (i) $\varphi_2 \in L(i)$ and (ii) ($\varphi_1 \in L(i)$ or $\varphi_1 \,\mathbf{R}\, \varphi_2 \in L(i+1)$).

If π is informative for ψ, the mapping L is called the *witness* for $\neg\psi$ in π.

The bad prefixes for a safety formula ψ can be used to classify formulas based on how well their counterexamples explain the violation of the property. A safety formula ψ is *intentionally* safe iff all the bad prefixes for ψ are informative. The formula $p\,\mathbf{U}\,q \vee \mathbf{G}\,p$ is intentionally safe. A formula ψ is *accidentally* safe iff every computation that violates ψ has an informative prefix. In other words, ψ can have bad prefixes which are not informative. Every computation is, however, guaranteed to have at least one informative prefix. For instance, $\mathbf{G}\ (p \Rightarrow (\mathbf{X}\,\mathbf{G}\,q \wedge \neg\mathbf{X}\,q))$ is accidentally safe. A safety formula ψ is *pathologically* safe if there is a computation that violates ψ and has no informative bad prefix. The formula $\mathbf{G}\,q \vee \mathbf{G}\,r \vee (\mathbf{G}\ (q \vee \mathbf{F}\,\mathbf{G}\,p) \wedge \mathbf{G}\ (r \vee \mathbf{F}\,\mathbf{G}\,\neg p))$ is pathologically safe [112]. A safety property for which all bad prefixes are not informative must contain some conflicting requirements or be vacuous in some sense. A pathological formula is always equivalent to some intentionally safe formula [112], but no procedure to rewrite a pathologically safe formula to an equivalent intentionally safe formula has been described in the literature.

3.1.2 Algorithms

Kupferman and Vardi [112] have showed that it is possible to construct an automaton, exponential in the size of the formula, which recognises all informative bad prefixes. An automaton recognising all informative bad prefixes can be used for model checking all non-pathological formulas but could potentially miss counterexamples for pathological safety formulas. Thus, if we want to avoid the doubly exponential construction to detect all bad prefixes, a method for determining whether a formula is pathological is needed. Sistla [150] has showed that the temporal operators $\mathbf{G}$, $\mathbf{R}$ and $\mathbf{X}$ form a *syntactically safe* subset of LTL when only positive combinations of these operators are allowed. Syntactically safe LTL formulas are always either intentionally safe or accidentally safe [112]. If all interesting properties could easily be expressed in this subset we would not need to check whether a formula is pathological. However, many interesting safety formulas are not syntactically safe. One example is the safety formula $\mathbf{G}\ ((q \wedge \neg r \wedge \mathbf{F}\,r) \Rightarrow \neg p\,\mathbf{U}\,r)$ from the specification pattern templates [50] that expresses "p is false between q and r".

Let ψ be an LTL formula and $\mathcal{A}^i_{\neg\psi}$ an FSA accepting all informative bad prefixes of ψ. $\mathcal{A}^i_{\neg\psi}$ can define a language over infinite strings, which we denote by $\mathcal{L}_\omega(\mathcal{A}^i_{\neg\psi})$, if we see $\mathcal{A}^i_{\neg\psi}$ as defining the good finite prefixes of the language. An infinite string belongs to $\mathcal{L}_\omega(\mathcal{A}^i_{\neg\psi})$ iff it has finite good prefix. A Büchi automaton accepting $\mathcal{L}_\omega(\mathcal{A}^i_{\neg\psi})$ can be obtained by interpreting $\mathcal{A}^i_{\neg\psi}$ as a Büchi automaton and adding a selfloop labelled with 2^{AP} to the unique accepting state of $\mathcal{A}^i_{\neg\psi}$. Then, ψ is not pathological iff $\mathcal{L}(\neg\psi) \subseteq \mathcal{L}_\omega(\mathcal{A}^i_{\neg\psi})$

In [**P1**] we presented a decision procedure for pathological formulas. The procedure is not complexity theoretically optimal, but seems to work fairly well in practice. Let $\bar{\mathcal{A}}^i_{\neg\psi}$ denote the complement of $\mathcal{A}^i_{\neg\psi}$ when it is seen as a Büchi automaton. Recall that deterministic Büchi automata can be complemented with a linear-time procedure. Given an LTL formulas ψ, the steps in the procedure are the following:

1. Construct a *deterministic* finite automaton $\mathcal{A}^i_{\neg\psi}$ for the informative bad prefixes of ψ.

2. Construct a Büchi automaton $\mathcal{B}_{\neg\psi}$ for $\neg\psi$.

3. Interpret $\mathcal{A}^i_{\neg\psi}$ as a Büchi automaton, and complement this deterministic Büchi automaton to produce the automaton $\bar{\mathcal{A}}^i_{\neg\psi}$.

4. Emptiness check $\bar{\mathcal{A}}^i_{\neg\psi} \times \mathcal{B}_{\neg\psi}$. Then, ψ is not pathological iff $\mathcal{L}(\bar{\mathcal{A}}^i_{\neg\psi} \times \mathcal{B}_{\neg\psi}) = \emptyset$.

As long as the size of $\mathcal{A}^i_{\neg\psi}$ remains reasonable this procedure works fairly well. An interesting property of the definition of pathological safety formulas is that any non-safety property will also be classified as pathological. No separate test is therefore needed. This can bee seen from the definition of safety languages. Any non-safety formula φ must have at least one infinite counterexample that does not have a bad prefix and can therefore not be captured by $\mathcal{L}_\omega(\mathcal{A}^i_{\neg\psi})$.

A prerequisite for checking if a formula is pathologically safe is the ability to construct a deterministic automaton that accepts all informative prefixes. Kupferman and Vardi [112] have derived an algorithm from the definition of informativeness. The algorithm produces a *reverse deterministic* finite automaton exponential in the size of the formula which is suitable for use in symbolic model checkers. An automaton is reverse deterministic if for a fixed letter $\sigma \in \Sigma$ each state has unique predecessor for that letter. This automaton is especially suitable for backwards symbolic reachability analysis [112].

In [**P1**] we presented an algorithm for an automaton accepting all informative prefixes of a safety property. The algorithm is a refinement the algorithm presented in [112]. Our version is more suitable for explicit state model checkers. The refined algorithm applies some of the standard reduction tricks already presented in [44] to produce small automata. Additionally, the implemented tool can determinise and minimise the produced reverse deterministic automaton. Minimisation was implemented after [**P1**] had been published. Determinising the automaton is usually a good idea for two reasons: the determinisation procedure usually makes the automaton smaller [**P1**], and deterministic automata tend to produce smaller product state spaces [60, **P1**, 145]. It is of course possible that determinising the automaton causes an exponential blow-up.

The algorithm as presented in [**P1**] contained a two small bugs, and additionally we have recently discovered a way to make the algorithm more efficient in the case where the LTL formula contains the next-operator. Therefore, we show the improved and correct construction here. The two small bugs that have been corrected are the following: In [**P1**] line 26 was placed between lines 24 and 25. This would cause incorrect results in the computation of S', since $sat(S, S', \psi)$ would return the wrong results. The second error was that the **if** on line 29 was missing, and thus all states would be added to the working set X without checking whether they had been encountered before. In order for the algorithm to terminate, the **if** is required.

We have recently discovered a way to optimise the algorithm when the formula under inspection contains the $\mathbf{X}$-operator. The optimisation does not require any changes in the pseudocode of the algorithm. Only the definitions of $rcl(\psi)$ and $sat(S, S', \psi)$ change slightly. The new definitions are given below.

As only the absolutely necessary notation is defined here, we refer to [**P1**] for a more detailed explanation of the construction. The construction is based on labelling the states of the automaton with subformulas. A label can be seen as corresponding to a proof requirement. If we reach a state with the empty label, there are no proof requirements left and we have found a witness for the formula. Symmetrically, all states labelled with the full formula are designated as initial states.

We define the restricted closure $rcl(\psi)$ of an LTL formula ψ (in positive normal form) as smallest set satisfying the following constraints.

- $\psi \in rcl(\psi)$.
- If $\psi_1 \mathbf{U} \psi_2 \in cl(\psi)$, then $\psi_1 \mathbf{U} \psi_2 \in rcl(\psi)$.
- If $\psi_1 \mathbf{R} \psi_2 \in cl(\psi)$, then $\psi_1 \mathbf{R} \psi_2 \in rcl(\psi)$.
- If $\mathbf{X} \psi_1 \in cl(\psi)$ then $\psi_1 \in rcl(\psi)$.

The intuition behind the restricted closure is that we only include the information which is necessary to remember for evaluating the temporal formulas. Because we minimise the information in the labels of the states, the missing information must be computed and expressed in the transitions from one state to another. Let S, S' be subsets of $cl(\psi)$. We define the function $sat(\psi, S, S')$ in the following way.

- $sat(\mathbf{true}, S, S') = \mathbf{true}$ and $sat(\mathbf{false}, S, S') = \mathbf{false}$.
- $sat(\psi, S, S') = \mathbf{true}$ if $\psi \in S$, otherwise $sat(\psi, S, S') = \mathbf{false}$.
- $\psi = \psi_1 \wedge \psi_2 : sat(\psi, S, S') = sat(\psi_1, S, S')$ and $sat(\psi_2, S, S')$.
- $\psi = \psi_1 \vee \psi_2 : sat(\psi, S, S') = sat(\psi_1, S, S')$ or $sat(\psi_2, S, S')$.
- $\psi = \mathbf{X} \psi_1 : sat(\psi, S, S') = sat(\psi_1, S', \emptyset)$.

The algorithm constructs the automaton by starting from an empty set of requirements and working backwards to construct all predecessors. For each state, we analyse which proof requirements can be discarded by moving to the current state from a potential predecessor. Each predecessor is the maximal set of subformulas compatible with moving to the current state on a certain atomic proposition.

The size of the automaton is in the worst case exponential in the number of temporal operators of the formula. This is better than $O(2^{|cl(\psi)|})$ that could be achieved with a direct implementation of the algorithm by Kupferman and Vardi [112].

Theorem 2 *The number of states of the automaton $\mathcal{A}^i_\psi$ bounded by $2^{|rcl(\psi)|}$. This bound is in $O(2^{|tf(\psi)|})$, where $tf(\psi)$ is the set of temporal subformulas of ψ.*

Constructing finite automata for model checking temporal logic has been considered in several papers. Many of these papers are concerned with the problem of monitoring software or runtime verification. Monitoring running

```
Input: A formula $\psi$ in positive normal form.
Output: A finite automaton $\mathcal{A}^i_\psi = \langle \Sigma, Q, \delta, Q_0, F \rangle$.
**proc** translate($\psi$)
$F := \{\emptyset\}; \Sigma := 2^{AP};$
$Q := X := \{\emptyset\};$
**while**($X \neq \emptyset$) **do**
    $S :=$"some set in $X$"; $X := X \setminus \{S\}$
    **for** each $\sigma \in 2^{AP}$ **do**
        $S' := \sigma;$
        **for** each $\varphi \in rcl(\psi)$ **do** //in increasing subformula order
            **switch**($\varphi$) **begin**
                case $\varphi = \psi_1 \vee \psi_2$:
                **if** ($sat(\psi_1, S', S)$ or $sat(\psi_2, S', S)$) **then** $S' := S' \cup \{\varphi\};$
                case $\varphi = \psi_1 \wedge \psi_2$:
                **if** ($sat(\psi_1, S', S)$ and $sat(\psi_2, S', S)$) **then** $S' := S' \cup \{\varphi\};$
                case $\varphi = \mathbf{X}\, \psi_1$:
                **if** ($\psi_1 \in S$) **then** $S' := S' \cup \{\varphi\};$
                case $\varphi = \psi_1 \,\mathbf{U}\, \psi_2$:
                **if** ($sat(\psi_2, S', S)$ or ($sat(\psi_1, S', S)$ and $\varphi \in S$))
                    **then** $S' := S' \cup \{\varphi\};$
                case $\varphi = \psi_1 \,\mathbf{R}\, \psi_2$:
                **if** ($sat(\psi_2, S', S)$ and ($sat(\psi_1, S', S)$ or $\varphi \in S$))
                    **then** $S' := S' \cup \{\varphi\};$
            **end**
        **od**
        **if** $\sigma \notin rcl(\psi)$ **then** $S' := S' \setminus \{\sigma\};$
        **if**($sat(\psi, S', \emptyset)$) **then** $Q_0 := Q_0 \cup \{S'\};$
        $\delta(S', \sigma) = \delta(S', \sigma) \cup \{S\};$
        **if** $S' \notin Q$ **then** $X := X \cup \{S'\}; Q := Q \cup \{S'\};$
    **od**
**od**
```

Figure 3.1: Algorithm for constructing a finite automaton for informative prefixes.

software either requires a deterministic automaton or on-the-fly determinisation of the automaton representing the illegal behaviours. Geilen [65] presents a tableau construction for monitoring bad and good prefixes of LTL formulas. The construction is essentially a forward version of the algorithm of Kupferman and Vardi [112] and has the same complexity. Havelund and Roşu [83] describe an algorithm for synthesising monitors for a past temporal logic. The method is similar to the history variables method presented in [12]. Both methods could be used to generate a deterministic finite automaton for a past temporal logic with the syntactic form $\mathbf{G}\,\psi$, where the temporal operators in ψ must be past operators. See Section 4 for an example of a temporal logic with past operators.

Sen et al. [146] describe a method based on coinduction to generate a minimal deterministic finite automaton for *all* bad prefixes. The intended target application is monitoring of software. The advantage of the method compared with the method described in [112] is that the method generates the optimal (potentially doubly exponential) automaton directly. In [112] an exponential Büchi automaton is generated as an intermediate step.

An automaton for informative bad prefixes can be used to detect minimal counterexamples for intentionally safe formulas. Since not all bad prefixes are detected, minimality for accidentally safe formulas is not guaranteed. However, it is questionable if non-informative counterexamples are useful for debugging purposes. By definition, the non-informative counterexamples are due to formulas that are malformed in the sense that they do not have only informative counterexamples. However, as long as there is no efficient procedure to transform pathological formulas to non-pathological formulas, completeness of a safety model checking procedure cannot be guaranteed without detecting all counterexamples. Understanding and minimising counterexamples is an area which has received a fair amount of attention lately [100, 135, 73, 148, 143].

3.2 MODEL CHECKING LIVENESS PROPERTIES OF MODULAR PETRI NETS

A popular way of describing systems is using Petri nets [136]. Petri nets can be seen as a generalisation of communicating automata. Concurrency can easily be described with Petri nets and synchronisation is described explicitly.

In their most basic form Petri nets have no concept of a module or a process that could be used to split a system description into parts. Petri nets describe a system as one monolithic object. However, splitting the description is convenient for designers and it has many benefits. Reasons often cited are that a modular description (i) matches the system design better, (ii) facilitates reuse of the design, (iii) enables reuse of analysis results and (iv) makes it is easier to get an overview of the system.

Modular Petri nets [26] extend Petri nets by adding a module/process structure to Petri nets. In modular Petri nets modules interact through transition fusion and shared places. We will however restrict our attention to nets using only transition fusion. Shared places can always be simulated by transition fusion [26]. This section presents an automata-theoretic method for model checking modular Petri nets.

3.2.1 Petri Nets and Modular Nets

This section presents the relevant notations and definitions.

Definition 3 *A Place/Transition net (PT-net) is a tuple* $N = (P, T, W, M_0)$ *where* P *is a finite set of places,* T *is a finite set of transitions such that* $P \cap T = \emptyset$*,* $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ *is the arc weight function and* $M_0 : P \to \mathbb{N}$ *is the initial marking.*

A marking is a multiset over P. For a transition $t \in T$ we identify $t^\bullet$ $({}^\bullet t)$ with the multiset given by $t^\bullet(p) = W(t,p)$ $({}^\bullet t(p) = W(p,t))$ for any $p \in P$.

A transition $t \in T$ is *enabled* in a marking M iff ${}^\bullet t \subseteq M$. In a marking M, an enabled transition t can *occur* resulting in the marking $M' = M - {}^\bullet t + t^\bullet$. This is denoted $M \xrightarrow{t} M'$. The notation can be generalised to a sequence $\sigma = t_1 t_2 \ldots t_n$ of transitions: $M_1 \xrightarrow{\sigma} M_{n+1}$ denotes that there exists a sequence of markings such that $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_{n+1}$. A marking in which no transition is enabled is called a *deadlocking* marking, while any marking that can be produced by the occurrence of a sequence of transitions from the initial marking is called *reachable*.

A modular Petri net describes a system as a collection of Petri nets. Synchronisation between the modules is accomplished using transition fusion.

Definition 4 *A modular PT-net is a tuple* $\mathcal{N} = (S, \mathit{TF})$ *where:*

- S *is a finite set of* modules*:*
  - *each module* $s \in S$ *is a PT-net* $s = (P_s, T_s, W_s, M_{0_s})$*,*
  - *the sets of nodes corresponding to different modules are pairwise disjoint, i.e., for all* $s_1, s_2 \in S : s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1}) \cap (P_{s_2} \cup T_{s_2}) = \emptyset$.
- *Let* $T = \bigcup_{s \in S} T_s$ *be the set of all transitions.* $\mathit{TF} \subseteq 2^T$ *is a finite set of* transition fusion sets *such that for all* $\mathit{tf} \in \mathit{TF}$ *we have that if* $t_i, t_j \in \mathit{tf}$ *and* $i \neq j$ *then* $t_i \in T_s \Rightarrow t_j \notin T_s$*. In other words, a module may contribute only one transition to a fusion transition but a transition can participate in several fusion transitions.*

A global marking M of a modular net is simply the union of the markings of the subnets, since the places of the subnets are disjoint. We call a transition not part of any transition fusion set an internal transition. By $M[[\sigma\rangle\rangle M'$, where $\sigma = t_0 t_1 t_2 \ldots t_n \mathit{tf}$, we denote that M' is reachable from M by a sequence of internal transitions $t_0 t_1 t_2 \ldots t_n$ followed by a fused transition tf. Christensen and Petrucci [26] have presented a method for computing a *synchronisation graph*. The graph shows the global behaviour of the system in terms of the fusion transitions.

Definition 5 *Let* $\mathcal{N} = (S, \mathit{TF}, M_0)$ *be a modular net with the initial marking* M_0*.* The synchronisation graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{v}_0)$ *of the net is defined inductively as follows:*

- $\mathbf{v}_0 = M_0 \in \mathbf{V}$.

- *If* $M \in \mathbf{V}$ *and* $M[[t_1t_2 \ldots t_n tf\rangle\rangle M''$ *then* $(M, tf, M'') \in \mathbf{E}$ *and* $M'' \in \mathbf{V}$.

- $\mathbf{V}$ *and* $\mathbf{E}$ *contain no other elements.*

The synchronisation graph has a few attractive features. It can in some cases be constructed even if the local components have infinite state spaces when considered in isolation [26]. Since the synchronisation graph only includes fusion transitions, it can be much smaller than the reachability graph representing the full behaviour. It is fairly straight-forward to compute interesting properties such as absence of deadlocks using the synchronisation graph, by checking some simple conditions local to the modules [26]. Mäkelä [123] has extended the basic results from standard modular Petri nets to nested modular high-level Petri nets. Nesting allows succinct description of hierarchy and can be very expressive when modelling large systems. Mäkela also describes an implementation of computing the synchronisation graph in the Petri net reachability analyser Maria [122].

It is possible to prove any LTL-X property using an adapted automata theoretic model checking algorithm [**P2**]. Essentially the model checking problem is reduced to an instance of tester verification, a topic which is discussed in Section 3.3. We say that a transition is *invisible* if it cannot affect the truth value of the atomic propositions of the given LTL formula. The presented method requires that all visible transitions are present in the synchronisation graph. Currently only a simple analysis is made to determine whether a transition is visible and the result is a safe over-approximation. A potential avenue of future work is computing more accurate over-approximations of visibility of transitions.

Constructing the synchronisation graph is similar to a few other methods for alleviating the state explosion problem. Katz and Miller [105] present a method for pruning invisible transitions from the state graph. The method produces a stuttering equivalent [117] state graph w.r.t. to the original state graph at the cost of an additional traversal of the original state graph. Since the reduced state graph is stuttering equivalent to the original state graph, it preserves all LTL-X properties. By applying partial order reduction methods and state space caching methods [69] the performance of the reduction method can be improved significantly. The use of state space caching avoids the overhead of ever having to remember the full state graph while the use of partial order reduction methods both reduces the memory overhead and improves the final result. A disadvantage of the method is that it is incompatible with on-the-fly model checking methods (see below).

Yorav [175] describes a method for path reduction in both sequential program and parallel programs. Again, the idea is based on eliminating all invisible transitions (w.r.t. the specification) from the state graph. The method works on the level of the control flow graph of the given program and produces a reduced control flow graph. The sequential case is extended to the parallel case by applying a modified path reduction to the individual processes which also preserves send and receive statements. The reduced program is stuttering bisimiliar with the original program and therefore preserves all CTL*-X properties. Kurshan et al. [116] present a method similar to Yorav's method for compressing invisible transitions. Their reasoning is explicitly based on partial order reduction methods for producing a reduced control flow graph. They argue that their method can reduce the state space more aggressively than Yorav's because they can use the theory for partial order reductions instead of only depending on static syntactic transformations.

Compared with the methods of Katz and Miller and Kurshan et al., and Yorav's path reduction method, there are some subtle differences in how model checking is performed in [**P2**]. The most notable difference is in how loops of invisible transitions are treated. All three previous methods can guarantee that all loops of invisible transitions have their counterparts in the reduced state graph. This allows them to use standard automata theoretic techniques for model checking the state graph. Loops of invisible transition are not preserved by the synchronisation graph. We could force all invisible transitions to appear in the synchronisation graph, but this would certainly eliminate any reduction in the state space we could gain by using it. We therefore choose to modify the standard automata theoretic procedure to use a weaker form of synchronisation and reduce the model checking problem to an instance of tester verification. An interesting topic for future work is whether the definition of the synchronisation graph could be modified to preserve loops of invisible transitions while allowing maximal reduction.

3.3 ON-THE FLY VERIFICATION USING TESTERS

An alternative approach to using Büchi automata for model checking and verification is using testers. Testers are a special form of automata that are structurally very similar to Büchi automata. In model checking using Büchi automata the system synchronises at each step with the Büchi automaton. When performing verification with a tester, a more general form of synchronisation is used. The synchronisation explicitly considers the visibility of transitions and a tester only synchronises with visible actions of the system under inspection. This allows testers to be used for verifying systems where a tighter form of synchronisation would reduce the efficiency of certain state space reduction methods such as net unfoldings [59] and modular analysis (see above).

To be able to express all relevant properties, while hampered by the looser than normal synchronisation with the system, the acceptance condition of testers is strictly more general than for Büchi automata. The stronger acceptance condition also allows deterministic testers to accept some languages that require non-deterministic Büchi automata. However, there are languages that cannot be accepted by a deterministic tester but can be captured a non-deterministic one [81].

Verification with techniques related to testers has been considered in several papers. Valmari [163] introduced the tester framework to allow efficient use of stubborn sets, a partial order reduction method. Esparza and Heljanko [59] borrow techniques from testers to facilitate model checking of Petri net unfoldings. As explained in Section 3.2, a similar method is used for model checking modular Petri nets in [**P2**]. Testers are also relevant for compositional verification techniques using compositional behavioural preorders like the NDFD preorder, which is the weakest compositional preorder and equivalence that preserves LTL-X properties [103]. Testers can be used to decide the CFFD behavioural preorder [86], a closely related preorder to NDFD.

3.3.1 Labelled Transition Systems and Testers

Testers are usually defined in the framework of *labelled transition systems* (LTSs). An LTS describes a system as an automaton with named actions.

Definition 6 *A* labelled transition system *is a quadruple* $P = (S, \Sigma, \Delta, s_0)$ *where* S *is the finite set of* states, Σ *is the finite set of* actions, $\Delta \subseteq S \times \Sigma \times S$ *is the set of* transitions *and* $s_0 \in S$ *is the initial state.*

In most cases a system is modelled as a collection of communicating LTSs. Communication is achieved by forcing the LTSs to synchronise on common actions. An LTS may perform an action independently of the other components in the system if the action does not appear in the alphabet of the other components.

Definition 7 *Let* $P_1 = (S_1, \Sigma_1, \Delta_1, s_{0_1}), \ldots, P_k = (S_k, \Sigma_k, \Delta_k, s_{0_k})$ *be LTSs for some* $k \geq 1$. *The* parallel composition *of* $P_1, P_2, \ldots, P_k$ *is the LTS* $P_1 \| P_2 \| \cdots \| P_k = (S, \Sigma, \Delta, s_0)$, *where*

- $S = S_1 \times \cdots \times S_k$;
- $\Sigma = \bigcup_{i=1}^{k} \Sigma_i$;
- *for all* $s = (s_1, s_2, \ldots, s_k), s' = (s'_1, s'_2, \ldots, s'_k) \in S$ *and* $a \in \Sigma$, $(s, a, s') \in \Delta$ *iff, for all* $1 \leq i \leq k$,
  - *if* $a \in \Sigma_i$, *then* $(s_i, a, s'_i) \in \Delta_i$ *and*
  - $s'_i = s_i$ *otherwise.*

We define the following notation.

Definition 8 *Let* $P = (S, \Sigma, \Delta, s_0)$ *be an LTS.*

- *For all* $s, s' \in S$ *and* $a \in \Sigma$, $s \text{---} a \longrightarrow s'$ *iff* $(s, a, s') \in \Delta$.
- *For all* $n \geq 0$, $s, s' \in S$ *and* $\sigma = a_1 a_2 \ldots a_n \in \Sigma^n$, $s \text{---} \sigma \longrightarrow s'$ *iff there exist states* $s_1, \ldots, s_{n+1} \in S$ *such that* $s_1 = s$, $s_{n+1} = s'$, *and* $s_i \text{---} a_i \longrightarrow s_{i+1}$ *holds for all* $1 \leq i \leq n$.

Properties of LTSs are described in terms of the possible sequences of transitions. In most cases we are only interested in a designated set $\Sigma_{vis} \subseteq \Sigma$ of *visible actions*. Similarly as in the automata-theoretic approach, an extended LTS called a tester is used to define the illegal behaviours. A tester is structurally similar to a Büchi automaton but has more general acceptance conditions. The synchronisation used when model checking with Büchi automata can simulated by setting $\Sigma_{vis} = \Sigma$.

Definition 9 *[163] Let* Σ_{vis} *be the set of visible actions of an LTS* $P_S = (S_S, \Sigma_S, \Delta_S, s_{0_S})$. *A* tester *(for* P_S*) is a tuple* $T = (P_T, S_R, S_D, S_L, S_\infty)$, *where*

- *$P_T = (S_T, \Sigma_T, \Delta_T, s_{0_T})$ is an LTS;*
- *$\Sigma_T = \Sigma_{vis} \cup \{\tau_T\}$, where τ_T is a new action unique to the tester (i.e., $\tau_T \notin \Sigma_S$);*
- *$S_R \cup S_D \cup S_L \cup S_\infty \subseteq S$;*
- *Δ_T contains no τ_T-loops;*
- *if $(s, \tau_T, s') \in \Delta_T$ for some $s, s' \in S_T$, then $s \notin S_D$.*

The sets S_R through S_∞ are called reject states, deadlock monitor states, livelock monitor states, *and* infinite trace monitor states, *respectively.*

The errors a tester can detect can be defined through the parallel composition of the system and the tester.

Definition 10 *[163] Let $P_T = (S_T, \Sigma_T, \Delta_T, s_{0_T})$ be an LTS associated with a tester for the LTS $P_S = (S_S, \Sigma_S, \Delta_S, s_{0_S})$, and let $(P_T \| P_S) = P = (S, \Sigma, \Delta, s_0)$ be the parallel composition of P_T and P_S.*

- *P_S has an illegal finite trace iff there exists a state $s = (s_T, s_S)$ reachable in P such that $s_T \in S_R$.*
- *P_S has an illegal stable failure iff there exists a state $s = (s_T, s_S)$ reachable in P such that $s_T \in S_D$ and s has no outgoing transition.*
- *P_S has an illegal divergence trace iff there exists a state $s_T \in S_L$, states $s_1 s_2 \ldots s_n$ $(n \geq 1)$ and actions $a_1, a_2 \ldots, a_n \in \Sigma_S \setminus \Sigma_{vis}$ such that (s_T, s_1) is reachable in P, and $((s_T, s_i), a_i, (s_T, s_{i+1})) \in \Delta$ for all $1 \leq i < n$ and $((s_T, s_n), a_n, (s_T, s_1)) \in \Delta$.*
- *P_S has an illegal infinite trace iff there exists a state $s_T \in S_\infty$, and a sequence of actions $\sigma = a_1 a_2 \ldots a_n \in (\Sigma_S \cup \{\tau_T\})^*$ $(n \geq 1)$ such that (s_T, s_S) is reachable in P, $(s_T, s_S) \xrightarrow{\;\sigma\;} (s_T, s_S)$ and $a_i \in \Sigma_{vis}$ for some $1 \leq i \leq n$.*

Illegal finite traces are a convenient way of specifying violations of safety properties. The illegal infinite traces are defined like Büchi automata acceptance and can e.g. be used to capture violation of fairness properties. Stable failures are mostly used to distinguish between legal and illegal deadlocking situations. The need for illegal divergence traces is a result of that not all actions are visible. Illegal divergence traces can capture situations where the system can still advance, but nothing visible occurs. For more detailed explanations please see [**P3**].

3.3.2 Algorithms

Several algorithms have been developed for finding errors in systems. Most algorithm are so called on-the-fly algorithms that detect errors while the state space is being constructed: there is no need to pre-compute the full state space. This is a desirable property because constructing the full state space can be very time consuming and we can still potentially find some errors

```
proc dfs($s$)
    $visited := visited \cup \{(s, 0)\}$;
    foreach successor $t$ of $s$ do
        if $(t, 0) \notin visited$ then dfs($t$) fi
    od
    if accepting($s$) then $seed := s$; ndfs($s$); fi
end
```

```
proc ndfs($s$)
    $visited := visited \cup \{(s, 1)\}$;
    foreach successor $t$ of $s$ do
        if $(t, 1) \notin visited$ then ndfs($t$)
        elsif $seed = t$ report "cycle found";
        fi
    od
end
```

Figure 3.2: Nested depth-first search algorithm (NDFS).

even though constructing the full state space is not within the available computational resources. Detecting illegal finite traces or illegal stable failures specified by testers can be easily reduced to the reachability of illegal states in the parallel composition of the system and the tester. Possible complete solutions include using standard depth-first search (DFS), breadth-first search (BFS), or any number algorithms which are guaranteed to visit all states in the parallel composition, including heuristic searches based on e.g. the A* algorithm [82]. Finding an illegal divergence trace entails finding a loop of invisible transitions when the tester is in a livelock monitor state. Holzmann [88] describes an algorithm for detecting non-progress cycles that can be easily adapted to finding illegal divergence traces. The algorithm searches for livelock monitor states in the parallel composition using DFS. A second DFS is started from all encountered livelock monitor states. The second DFS only traverses invisible transitions and tries to complete a loop returning to the start state. Valmari [163] presents an algorithm that can find illegal divergence traces in one pass of the state space of the parallel composition. The algorithm maintains a frontier of states from where it starts a DFS that only traverses invisible transitions and livelock monitor states. States that do not qualify are added to the frontier. If the DFS finds a loop a livelock error is reported.

Finding an illegal infinite trace requires finding a loop in the state space that satisfies two independent conditions: i) at least one state on the loop must be an infinite trace monitor state and ii) at least one of the transitions on the loop must be visible. Solving this problem can be reduced to performing an emptiness check of a generalised Büchi automaton with two accepting sets. To solve the case with only one acceptance condition the most common solution is the nested depth-first search algorithm [42]. The algorithm is shown in Figure 3.2.

Nested depth-first search (NDFS) works by searching for accepting states in a depth-first manner. When the top-level DFS backtracks from an accepting state, a second DFS is started (line 6). The second DFS works in its own copy of the state space and reports an accepting cycle (line 5) if it encounters the seed state, i.e. the state where the second search was started. Godefroid and Holzmann [68] showed that the second copy of the state space can be maintained by using only two additional bits for each stored state. Changing line 5 in the second DFS to report a cycle if a state on the DFS stack of the dfs(s) procedure can allow earlier detection counterexamples at the cost of storing one additional bit per state [90]. A version of the algorithm where one of the three bits can be eliminated is described in [36].

The tester verification problem requires solving the generalised Büchi emptiness problem with two accepting sets as a subproblem. One solution to the generalised Büchi emptiness problem can be obtained by reducing the generalised Büchi automaton to a standard Büchi automaton [55, 42]. Given a generalised Büchi automaton $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \mathcal{F} \rangle$, where $\mathcal{F} = \{F_1, \ldots, F_k\}$ the equivalent standard automaton $\mathcal{A}_s = \langle \Sigma, Q_s, \delta_s, Q_0^s, F_s \rangle$ is defined by the following.

- $Q_s = Q \times \{1, 2, \ldots, k\}$
- $Q_0^s = Q_0 \times \{1\}$
- $F_s = F_1 \times \{1\}$
- $$\delta_s((q,i),a) = \begin{cases} \delta(q,a) \times \{i\} & \text{if } q \notin F_i \\ \delta(q,a) \times \{1 + (i \; mod \; k)\} & \text{if } q \in F_i \end{cases}$$

The basic idea of the construction is to “unfold” the generalised Büchi automaton into k interconnected copies where moving from a certain copy i to the next copy $1 + (i \; mod \; k)$ is only allowed when an accepting state in F_i is encountered. Thus, if we encounter states in F_s infinitely often we must have seen states in all $F_1, F_2, \ldots F_k$ infinitely often. The unfolded automaton is k times bigger than the original and accepts exactly the same language as the original automaton. NDFS algorithms which unfold the generalised Büchi automaton on-the-fly while exploring the state space are referred to as generalised NDFS algorithms.

The solution to the tester verification problem presented in [**P3**] is based on a generalised NDFS combined with Valmari’s algorithm [163]. The two key observations of the paper are that a generalised NDFS is actually required (i.e. two Büchi acceptance sets are needed), and that the correctness of NDFS is not dependent on the search order of the second search. Only in the the top-level search is the post-order calling of the nested search critical. This observation allows the second DFS to be replaced by a modified version of Valmari’s algorithm. In the worst case the algorithm traverses the original product state space four times and uses seven bits of additional memory per state [**P3**]. An important property of the algorithm is that is works on-the-fly.

There are several papers which have studied different ways of improving the basic NDFS for solving the Büchi emptiness problem. We survey the most important ones and discuss if the suggested improvements could adapted to the solution presented in [**P3**]. Brim et al. [20] present a modification of the basic NDFS where states are randomly removed from the set of visited states to conserve memory. The method is somewhat similar to state space caching [69] where the basic idea is also not to store all visited states. Both methods result in states potentially being re-explored several times. While a DFS is guaranteed to terminate if all states on the stack are in the *visited* set, our algorithm does not trivially have this property since the second search does not explore states in a depth-first manner. Edelkamp et al. [51] analyse the SCC structure of the given Büchi automaton to avoid calling the second DFS. The Büchi automaton is partitioned into components according to if all cycles or no cycles in a component go through an accepting state. In the third category of components only some cycles are accepting. Properties with Büchi automata that do not have components of the third kind can be verified in one pass of the state space using a modified simple DFS. The problem with adapting this optimisation to the tester verification problem is lifting the SCC partitioning to the generalised acceptance condition of testers, which also depends on the visibility of transitions of the system.

Gastin et al. [63] improve the standard NDFS by detecting some counterexamples using only the top-level DFS and avoiding unnecessary revisits of states by the second search in NDFS. Schwoon and Esparza [144] analyse the improvements of Gastin et al. [63] and slightly generalise their result. They present a more memory efficient way to implement the optimisations using only two bits of additional memory. By adapting the optimisations of [144] to the top-level DFS of the algorithm of [**P3**], it is possible to detect some illegal infinite traces earlier. This could be accomplished by adding a check whether the top-level search has encountered an infinite monitor state on the search stack reached by a visible transition. How much could be gained by this optimisation is unclear.

Tauriainen [157] has presented a generalised NDFS solving the generalised Büchi emptiness problem in k+1 passes of the state space. Compared with $2k$ passes for an algorithm that unfolds the generalised automaton and applies the standard NDFS algorithm this is a significant improvement. If the algorithm could be amended to also detect illegal divergence traces essentially without overhead, as has been done for the NDFS algorithm in [**P3**], the resulting algorithm could find all violations specified by a tester in three passes of the state space. This is an open problem.

When one of the algorithms above can detect a counterexample in one pass of the state space, the underlying reason is the structure of the given property. Bloem et al. [19] show that if the Büchi automata representing the negation of the property is *weak*, a simple DFS can solve the emptiness problem. A Büchi automaton is weak if the states Q of the automaton can be partitioned such that the partition forms a partial order w.r.t. the transition function of the automaton and all states in each partition are either accepting or non-accepting. Schneider [141] has shown that large classes of properties can be captured by weak automata. According to [25] approximately 95% of the properties in the specification pattern database [50] can be captured by weak Büchi automata.

It is unclear what the relation between weak Büchi automata and testers with only livelock monitor states is. However, based on the fact that weak automata can be characterised by the acceptance condition $\mathbf{F}\,\mathbf{G}\,p$ [141] we conjecture that all weak automata that accept stuttering invariant languages have an equivalent tester with only livelock monitor states. As mentioned above, Valmari's algorithm [163] can find any violation of these testers in one pass of the state space.

Instead of applying a NDFS-based approach to solve the tester verification problem, one could apply an approach based on computing the SCCs in the product state space. In [**P3**] we show how the tester verification problem can be reduced to a Streett automata emptiness check. *Streett automata* are automata on infinite words with an acceptance condition similar to strong fairness (see e.g. [160]). Compared with a generalised Büchi automaton, a Streett automaton replaces the family of accepting sets $\mathcal{F}$ with a family of acceptance pairs $\Omega = \{(L_1, U_1), \ldots, (U_k, L_k)\}$. A Streett automaton accepts a run ρ if $\bigwedge_{i=1}^{k} (\mathit{inf}(\rho) \cap L_i \neq \emptyset \Rightarrow \mathit{inf}(\rho) \cap U_i \neq \emptyset)$. Emptiness algorithms for Streett automata are usually based on computing the SCCs of the automaton. Although emptiness checking algorithms for Streett automata have non-linear worst case time behaviour [54, 87, 119], the resulting algorithm is a linear-time algorithm as it uses only two acceptance pairs for the reduction to Streett Automata. Other SCC-based single pass linear-time algorithms [44, 66, 144] designed for emptiness checking of Büchi automata cannot be trivially generalised to solve the tester verification problem. The key problem that needs to be solved is the detection of illegal divergence traces.

4 BOUNDED MODEL CHECKING

Bounded model checking [14] was introduced as an alternative to BDDs to implement symbolic model checking. The basic idea behind bounded model checking is to restrict the general model checking problem to a bounded problem. Instead of asking whether the system M violates the property ψ, we ask whether the system M has any counterexample of length k to ψ. This bounded problem is mapped to SAT in order to obtain the benefits of a symbolic representation of the state space. In other words, a Boolean formula $|[M, \neg\psi, k]|$ is generated which is satisfiable iff M has a counterexample of length k to ψ. This can be checked with a SAT solver.

The key insight behind BMC for LTL is that the infinite paths in the system that are models for LTL formulas can be captured by a finite path in two ways: either the finite path represents all its infinite extensions or the finite path loops and it in fact captures the behaviour of an infinite path. Let $\pi = s_0 s_1 s_2 \ldots$ be an infinite path. We say that π is a (k,l)-loop if $\pi = (s_0 s_1 \ldots s_{l-1})(s_l \ldots s_k)^\omega$. We write $p(\pi) = k - l + 1$ for the *period* of π. For finite state systems all LTL counterexamples can be captured by a (k,l)-loop [168].

In BMC the transition relation $T(s,s')$ is represented symbolically as a propositional formula, where the states s, s' are modelled as bit vectors. To capture the finite paths of length k, we unroll the transition relation k times and get the following Boolean formula:

$$|[M]|_k := I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i).$$

Here $I(s)$ is the initial state predicate and $T(s,s')$ a total transition relation. Since only counterexamples to the given LTL formula ψ should be accepted, additional constraints must be generated to restrict the models of the Boolean formula. If we denote the formula constraints by $|[\neg\psi]|_k$, the Boolean formula $|[M, \neg\psi, k]| := |[M]|_k \wedge |[\neg\psi]|_k$ is satisfiable iff M has a counterexample of length k to ψ.

Compared with using BDDs to implement symbolic model checking, BMC has a few advantages. BMC can leverage the impressive gains that have been achieved in SAT solver technology in the recent years. SAT solvers are comparatively memory efficient and can be instructed to use only a linear amount of memory w.r.t. size of the given Boolean formula. However, this restricts the use of the very efficient learning heuristics. The increase in efficiency of the solvers can directly be translated to more effective BMC. Another important advantage of BMC is that the counterexamples produced by most BMC encodings are minimal. Producing short counterexamples using BDDs is a fairly involved process [35] and minimality is seldom guaranteed. In many cases producing the counterexample consumes more resources than answering the model checking query [35]. However, recently a BDD model checking procedure [143] based on the BMC encoding of [**P5**] was presented that provably produces minimal counterexamples. The method appears to consume more memory than standard BDD model checkers, but can in some cases be faster. Another advantage of BMC is that Boolean formulas are a more compact encoding than BDDs for many Boolean functions: there are Boolean functions whose BDDs are exponential in the number of propositional variables [22]. BMC is not always more space efficient than using BDDs. For a simple binary counter system an exponential number of unrollings of the transition relation is required before the system loops and we can be sure that the whole behaviour has been covered. Thus, even though the SAT solver is memory efficient, we may end up using an exponential amount of memory w.r.t. system description. However, unlike for BDDs the increase in memory consumption for BMC is predictable w.r.t. increases in the bound k.

From its inception BMC has been predominantly seen as an efficient method for finding bugs. BDD-based methods have had the advantage of being complete and thus being able to prove that no counterexample exists. To prove that a system has no counterexamples for a given property with BMC, we must prove that no counterexample can be longer than a certain bound, the *completeness threshold*, and prove that there are no shorter counterexamples. Determining the completeness threshold is challenging. However, several methods have been developed in the recent years which can be used to achieve completeness with BMC.

An efficient method of finding small completeness thresholds for invariant properties is using induction. Sheeran et al. [147] present an inductive scheme for invariants. They show that invariants can be proven by strengthening induction: no path of length k may break the invariant and there is no loop-free path, which does not visit an initial state, of length $k + 1$. The approach can be formulated in automata-theoretic terms: A deterministic FSA for the property $\mathbf{F}\,\neg p$ (p is the invariant) is synchronised with the system. We stop when either a counterexample is found or we cannot increase the bound without closing a loop in the product system. The initial state optimisation of [147] is the only feature missing from this formulation.

The longest initialised loop-free path in the state graph is called the *recurrence diameter*, and for a bound k a straightforward encoding of the predicate required to express this is of the size $O(k^2)$. The recurrence diameter can be used as an upperbound for the completeness threshold when proving invariants. Kroening and Strichman [110] show that the size of this loop-free predicate can be optimised to $O(k \log^2 k)$ using sorting networks. They also suggest ways to leave out state bits from the loop-free predicate to improve efficiency while maintaining completeness. The benefits of having a smaller predicate are two-fold: a smaller predicate is easier to manage for the SAT solver and with fewer state variables we can prove properties at shallower depths because the system loops earlier.

The inductive method can easily be generalised to safety properties (e.g. using the method described in 3.1), and by using the liveness-to-safety transformation presented in [142] generalised to general LTL properties. The liveness-to-safety transformation doubles the number of state variables in the model. This increases the size of the already prohibitively large loop-free predicate. In addition, practical experience seems to indicate that already model checking general safety properties using induction is challenging [7]. Simply synchronising an FSA representing a safety property with the system to model check safety properties does not scale well, and forces model checkers to go deeper than the current capacity of SAT solvers. One reason is the non-determinism in the FSA representing the property [7]. Perhaps specifications using deterministic FSAs could be treated more efficiently [**P1**].

Two papers that consider strengthening of induction without always doing deeper BMC queries, which is expensive, are [47, 7]. In [47] the inductive method of [147] is generalised to an induction scheme based on simulations. Inductive invariants are automatically strengthened from failed induction proofs using a procedure based on existential quantification. Since existential quantification is resource intensive, a method for quantifying on demand is developed. Another approach is presented in [7]. They develop a methodology for flexible manual strengthening of induction. The key idea idea is to make the induction scheme part of the specification to allow a high degree of control of the induction process. Counterexamples produced by the model checker aid the designer in choosing new invariants.

Finding a completeness threshold for general LTL properties has proven fairly challenging. Clarke et al. [37] show how the completeness threshold can be computed for general LTL properties by computing the recurrence diameter of the product of the system and a Büchi automaton representing the negation of the property. Awedh and Somenzi [8] apply the same approach, but they use a refined method for calculating the completeness threshold. Both papers have the problem that they use an explicit representation of Büchi automata in their implementations. Thus, they potentially use an exponential number of state bits in the size of the formula to represent the Büchi automaton. The Büchi automaton is obtained from a generalised Büchi automaton using a procedure similar to the one presented in Section 3.3. These Büchi automata cannot capture minimal counterexamples of the LTL properties [8], and they might therefore have to proceed deeper to prove properties than methods based on generalised Büchi automata.

A different approach to proving completeness is taken by McMillan [127]. He uses interpolants derived from unsatisfiability proofs of BMC counterexample queries to over-approximate symbolic reachability. The deeper the BMC query is, the more exact the over-approximation is. The method is complete and can be extended to LTL model checking through the liveness to safety transformation [142]. Although the method can in many cases converge more quickly than the recurrence diameter, which is the relevant bound for most other methods, the unsatisfiability proofs can be of exponential size and cause a blow-up.

4.1 A NEW ENCODING FOR BMC

One of the key factors affecting the efficiency of BMC is the size of encoding $|[M, \psi, k]|$. If the encoding produces unnecessarily large formulas the solver will be quickly overwhelmed and we may not be able to proceed deep enough to find all violations to the specification in the design.

In [**P4**] we presented a BMC encoding for LTL which is linear in k that outperformed previous encodings. We briefly review our encoding for bounded model checking LTL. It consists of three types of constraints: model constraints, loop constraints and LTL constraints. *Model constraints* encode legal initialised finite paths of the model M of length k:

$$|[M]|_k := I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i),$$

where $I(s)$ is the initial state predicate and $T(s, s')$ is a total transition relation. The *loop constraints* are used to non-deterministically select loops. We introduce k fresh *loop selector variables* $l_1, \ldots, l_k$ which select where the path loops. At most one loop selector variable is allowed to be true. If l_i is true then $s_{i-1} = s_k$. In this case the LTL constraints treat the bounded path as a (k, i)-loop. If no loop selector variable is true the LTL constraints treat the path as a simple path without a loop. This optimisation can allow earlier detection of some counterexamples (informative safety counterexamples to be exact). This is accomplished with the following constraints:

$$\begin{aligned}
|[\mathit{LoopConstraints}]|_k &\Leftrightarrow \mathit{Loop}_k \wedge \mathit{AtMostOne}_k.\\
\mathit{Loop}_k &\Leftrightarrow \textstyle\bigwedge_{i=1}^{k} \left(l_i \Rightarrow (s_{i-1} = s_k)\right).\\
\mathit{AtMostOne}_k &\Leftrightarrow \textstyle\bigwedge_{i=1}^{k} \left(\mathit{InLoop}_{i-1} \Rightarrow \neg l_i\right).\\
\mathit{InLoop}_0 &\Leftrightarrow \bot.\\
\mathit{InLoop}_i &\Leftrightarrow \mathit{InLoop}_{i-1} \vee l_i, \text{ where } 0 < i \leq k.
\end{aligned}$$

InLoop_i is true if the position i is in loop part of the path. The loop selector variables indicate where the bounded path loops and select a (k, l)-loop from the model. Finally, *LTL constraints* check if the bounded path defined by the model constraints and loop constraints is a model of the LTL formula. The LTL encoding utilises the fact that for (k, l)-loops the semantics of CTL and LTL coincide [112, 158]. Essentially, the encoding can be seen as a CTL model checker for lasso-shaped Kripke structures based on using the least and greatest fixpoint characterisations of $\mathbf{U}$ and $\mathbf{R}$. The computation of the fixpoints for $\mathbf{U}$ and $\mathbf{R}$ is done in two parts. The auxiliary translation $\langle\langle\cdot\rangle\rangle$ computes an over-approximation for greatest fixpoints and an under-approximation for least fixpoints. The approximations are refined to exact values by $|[\cdot]|$.

The auxiliary translation $\langle\langle\cdot\rangle\rangle$ under-approximates $\psi_1 \,\mathbf{U}\, \psi_2$-formulas by assuming that ψ_2 does not hold beyond the end point. Thus, the recursive characterisation of $\mathbf{U}$ will evaluate to true only in states when ψ_2 holds in a future state. Conversely, $\psi_1 \,\mathbf{R}\, \psi_2$ is over-approximated by assuming that $\psi_1 \,\mathbf{R}\, \psi_2$ *holds* beyond the end point. Both of these approximations are exact at $i = l$ because of the simple looping structure of the models.

φ	$0 \leq i < k$	$i = k$
$\vert[p]\vert_i$	p_i	p_i
$\vert[\neg p]\vert_i$	$\neg p_i$	$\neg p_i$
$\vert[\psi_1 \wedge \psi_2]\vert_i$	$\vert[\psi_1]\vert_i \wedge \vert[\psi_2]\vert_i$	$\vert[\psi_1]\vert_i \wedge \vert[\psi_2]\vert_i$
$\vert[\psi_1 \vee \psi_2]\vert_i$	$\vert[\psi_1]\vert_i \vee \vert[\psi_2]\vert_i$	$\vert[\psi_1]\vert_i \vee \vert[\psi_2]\vert_i$
$\vert[\mathbf{X}\, \psi_1]\vert_i$	$\vert[\psi_1]\vert_{i+1}$	$\bigvee_{j=1}^{k} \left(l_j \wedge \vert[\psi_1]\vert_j\right)$
$\vert[\psi_1 \,\mathbf{U}\, \psi_2]\vert_i$	$\vert[\psi_2]\vert_i \vee \left(\vert[\psi_1]\vert_i \wedge \vert[\psi_1 \,\mathbf{U}\, \psi_2]\vert_{i+1}\right)$	$\vert[\psi_2]\vert_i \vee \left(\vert[\psi_1]\vert_i \wedge \left(\bigvee_{j=1}^{k} \left(l_j \wedge \langle\langle\psi_1 \,\mathbf{U}\, \psi_2\rangle\rangle_j\right)\right)\right)$
$\vert[\psi_1 \,\mathbf{R}\, \psi_2]\vert_i$	$\vert[\psi_2]\vert_i \wedge \left(\vert[\psi_1]\vert_i \vee \vert[\psi_1 \,\mathbf{R}\, \psi_2]\vert_{i+1}\right)$	$\vert[\psi_2]\vert_i \wedge \left(\vert[\psi_1]\vert_i \vee \left(\bigvee_{j=1}^{k} \left(l_j \wedge \langle\langle\psi_1 \,\mathbf{R}\, \psi_2\rangle\rangle_j\right)\right)\right)$
$\langle\langle\psi_1 \,\mathbf{U}\, \psi_2\rangle\rangle_i$	$\vert[\psi_2]\vert_i \vee \left(\vert[\psi_1]\vert_i \wedge \langle\langle\psi_1 \,\mathbf{U}\, \psi_2\rangle\rangle_{i+1}\right)$	$\vert[\psi_2]\vert_i$
$\langle\langle\psi_1 \,\mathbf{R}\, \psi_2\rangle\rangle_i$	$\vert[\psi_2]\vert_i \wedge \left(\vert[\psi_1]\vert_i \vee \langle\langle\psi_1 \,\mathbf{R}\, \psi_2\rangle\rangle_{i+1}\right)$	$\vert[\psi_2]\vert_i$

The conjunction of these three sets of constraints forms the full encoding of the bounded model checking problem into SAT:

$$|[M, \varphi, k]| := |[M]|_k \wedge |[LoopConstraints]|_k \wedge |[\varphi]|_0.$$

The LTL formula φ has a witness in M that can represented by a looping path of length k iff the encoding is satisfiable [**P4**].

The encoding has a few desirable properties of which the most important one is that when the encoding is seen as a Boolean circuit the size of the generated formula is $O(|I| + k \cdot (|T| + |\psi|)$. The encoding also has a unique model property in the following sense: if the (k, l)-loop is given (i.e. the computation π and l_i variables are fixed), the Boolean circuit representing the LTL encoding has no free variables and thus uniquely decides if the formula holds. In addition, if loop selector variables, atomic propositions and their negations are seen as inputs to the circuit, the circuit for the LTL encoding $|[\psi]|_0$ is monotonic.

The encoding is also fairly simple to describe and understand. Implementing the encoding is straightforward and its simple structure should make it possible to adapt the encoding to an incremental setting [171, 154, 52]. In incremental bounded model checking the efficiency of the SAT solver is improved by exploiting the similarities between two consecutive instances of BMC: the solver does not need to recompute everything if it can learn from previous instances.

The original encoding [14] and its improved version [29] both result in formulas that are at least quadratic w.r.t. k. Frish et al. [62] have presented an alternative encoding based on normal forms for LTL. The so called fixpoint encoding is more efficient than previous attempts, but it produces formulas that are non-linear w.r.t. k [**P4**]. An improved version of the fixpoint encoding, which includes a generalisation to LTL with past operators, is linear w.r.t. k [30]. The normal form used in the fixpoint encoding [62] is similar to tableau methods for constructing a symbolic Büchi automaton $\mathcal{A}_\psi$ representing an LTL formula ψ. It is also possible to do BMC by applying the automata theoretic approach and symbolically encode a product system $M \times \mathcal{A}_{\neg\psi}$ [46, 37]. BMC is performed by searching for fair loops [46, 37] in the product system. This approach does not produce a linear sized formula $|[M, \psi, k]|$ unless the search for fair loops is encoded with an improved encoding such as [29, **P4**]. Since this method only searches for looping counterexamples, it must sometimes go deeper than other methods also accepting simple paths as counterexamples.

The new encoding also has similarities with symbolic Büchi automata for LTL. If we consider the symbolic automata encoding of [33], the basic recursion used for the temporal operators is the same. The symbolic Büchi automaton uses fairness constraints to ensure that eventualities are not disregarded while our auxiliary translation essentially performs the same function. In the case of the Büchi automaton fairness constraints are only generated for $\mathbf{U}$-operators and, indeed, with small modifications the auxiliary translation could also be eliminated for $\mathbf{R}$-operators from our encoding. The reason is that the release operator $\psi_1 \mathbf{R} \psi_2$ is a so called weak temporal operator and does not require that ψ_1 must eventually hold at some point. However, with this optimisation the encoding would loose its unique model property. Evaluating the impact the optimisation and the exact implementation details are left for further work.

The relation between symbolic Büchi automata and BMC encodings is something that should be investigated further. Schuppan and Biere [143] have initiated work on this but there are still unanswered questions. For instance, could the optimisations for symbolic Büchi automata construction discussed in [141] be applied in the context of BMC? The relationship between weak alternating automata and the fixpoint encoding [62] has been investigated in [149]. A natural question is could optimisations developed for alternating automata [64, 156] be applied in the BMC context.

4.2 LTL WITH PAST

LTL can be extended with past temporal modalities, i.e. temporal operators that refer to the truth of subformulas in the past. The resulting logic is usually referred to as PLTL. Although the past operators do not add expressive power to LTL [151], it has been argued that PLTL is easier and more intuitive to use [120]. Some of this ease-of-use probably stems from the fact that PLTL is exponentially more succinct than LTL [118]. However, despite the succinctness gap between PLTL and LTL, the model checking problem is PSPACE-complete for both [151]. Practical model checking algorithms should be specially made for PLTL without going via a translation to LTL. Extending LTL with past operators has proven useful in compositional model checking [120], requirements engineering [165], runtime verification [83], and bounded model checking [11, 30].

PLTL has six new operators compared with LTL. The previous-operators $\mathbf{Y}\,\psi, \mathbf{Z}\,\psi$ state that ψ was true in the previous state but differ in their treatment of the initial state. The two other unary operators once $\mathbf{O}\,\psi$ and historically $\mathbf{H}\,\psi$ express that ψ was true once in the past and always in the past respectively. The since-operator $\psi_1\,\mathbf{S}\,\psi_2$ requires that ψ_2 was true once in the past and that ψ_1 has been true since. The dual of the since-operator is the trigger-operator $\psi_1\,\mathbf{T}\,\psi_2$. It states that ψ_2 must have been true from the point onward where ψ_1 was true. The requirement on ψ_1 is weak in the sense that if ψ_2 was always true in the past the trigger-operator is satisfied. We define the semantics of PLTL by extending the formal semantics of LTL. Only the semantics for the new operators are given.

$$\begin{array}{lll}
\pi^i \models \mathbf{Y}\,\psi & \Leftrightarrow & i > 0 \text{ and } \pi^{i-1} \models \psi. \\
\pi^i \models \mathbf{Z}\,\psi & \Leftrightarrow & i = 0 \text{ or } \pi^{i-1} \models \psi. \\
\pi^i \models \mathbf{O}\,\psi & \Leftrightarrow & \pi^j \models \psi \text{ for some } 0 \le j \le i. \\
\pi^i \models \mathbf{H}\,\psi & \Leftrightarrow & \pi^j \models \psi \text{ for all } 0 \le j \le i. \\
\pi^i \models \psi_1\,\mathbf{S}\,\psi_2 & \Leftrightarrow & \exists\, 0 \le j \le i : \pi^j \models \psi_2 \text{ and } \pi^n \models \psi_1 \text{ for all } j < n \le i. \\
\pi^i \models \psi_1\,\mathbf{T}\,\psi_2 & \Leftrightarrow & \forall\, 0 \le j \le i : \pi^j \models \psi_2 \text{ or } \pi^n \models \psi_1 \text{ for some } j < n \le i.
\end{array}$$

We denote by $\delta(\psi)$ the maximum nesting depth of past formulas in the PLTL formula ψ.

A simple example of a PLTL formula is $\mathbf{Z}\,\bot$, which is only true in the first state of a model. A more complex example is the formula $\mathbf{G}\,\mathbf{F}\ (p\,\mathbf{T}\,\neg q)$ that accepts models where $p \wedge \neg q$ is true infinitely often or q becomes false when

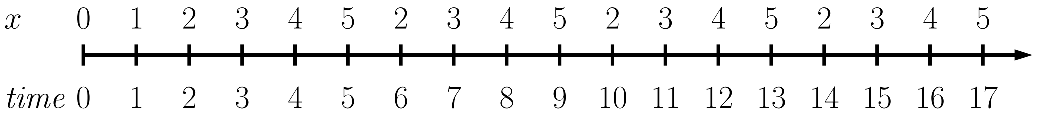
x 0 1 2 3 4 5 2 3 4 5 2 3 4 5 2 3 4 5
time 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Figure 4.1: A simple incrementing counter that resets to two when the counter reaches $x = 5$.

p is true at some point and remains false.

If we only consider the case of non-looping counterexamples, constructing a BMC encoding for PLTL is fairly easy. Since past operators look backwards, one simply has to examine the finite past of the current position i in the simple path $\pi = s_0 s_1 \dots s_k$ under consideration. If we consider looping counterexamples the situation becomes more complex. PLTL formulas cannot be evaluated as easily for the states inside the loop. A simple counter system (adapted from [11]) suffices to demonstrate the point. The counter increments a variable x until it reaches $x = 5$, at which point the variable is reset to $x = 2$. The situation is depicted in Figure 4.1. The system has a single execution $012(3452)^\omega$ that corresponds to a $(6, 3)$-loop. The formula $\mathbf{O}\ (x = 3 \wedge \mathbf{O}\, x = 4)$ holds for the first time at $i = 7$. Specifically it does not hold at $i = 3$ which is the equivalent state to $i = 7$ w.r.t. the $(6, 3)$-loop. The conclusion is that PLTL can distinguish between different unrollings of a (k, l)-loop. We formalise this intuition below.

Consider a (k, l)-loop $\pi = s_0 s_1 \dots s_k$. We define that a time point $i \geq 0$ in π belongs to the d-unrolling of the loop iff $d \geq 0$ is the smallest integer such that $i < l + ((d + 1) \cdot p(\pi))$. A result independently discovered by [118] and [11] shows that a PLTL formula can distinguish between different d-unrollings of a (k, l)-loop up to the past depth of the formula.

Proposition 1 *Let π be a (k, l)-loop and ψ a PLTL formula. For all $i \geq l + p(\pi) \cdot \delta(\psi)$, $\pi^i \models \psi$ iff $\pi^{i+p(\pi)} \models \psi$. Specifically, if the time point i belongs to a d-unrolling such that $d \geq \delta(\psi)$ we have that $\pi^i \models \psi$ iff $\pi^j \models \psi$ where $j = i - ((d - \delta(\psi)) \cdot p(\pi))$.*

Proof: The proof proceeds as an induction on the structure of the formula. For pure LTL formulas the result follows directly from the fact that the suffixes of π starting at i and j are identical. Let us consider the case $\psi = \psi_1\, \mathbf{S}\, \psi_2$. We start by proving the '⇒' direction. Assume that $\pi^i \models \psi$. By the semantics of $\mathbf{S}$ there exists an $m \leq i$ such that $\pi^m \models \psi_2$ and for all $m < n \leq i$ we have that $\pi^n \models \psi_1$. Applying the induction hypothesis we can infer that if $m \geq i - p(\pi)$, then $\pi^{m+p(\pi)} \models \psi_2$ and for all $m + p(\pi) < m' \leq i + p(\pi)$ we can conclude that $\pi^{m'} \models \psi_1$. Combining these we can deduce that $\pi^{i+p(\pi)} \models \psi$. On the other hand if $m < i - p(\pi)$ the semantics of the since-operator guarantees that $\pi^n \models \psi_1$ for $n = i - p(\pi), \dots, i$. By the induction hypothesis $\pi^n \models \psi_1$ also for $n = i, \dots, i + p(\pi)$. Consequently $\pi^{i+p(\pi)} \models \psi$.

To prove the other direction '⇐' we start by assuming that $\pi^{i+p(\pi)} \models \psi$. By the semantics we can again conclude that there is an $m \leq i + p(\pi)$ such that $\pi^m \models \psi_2$ and for all $m < n \leq i + p(\pi)$ we have that $\pi^n \models \psi_1$. If $m \leq i$ we can directly infer that $\pi^i \models \psi$. In the case $m > i$, we can apply the induction hypothesis and see that $\pi^{m-p(\pi)} \models \psi_2$ and $\pi^n \models \psi_1$ for all

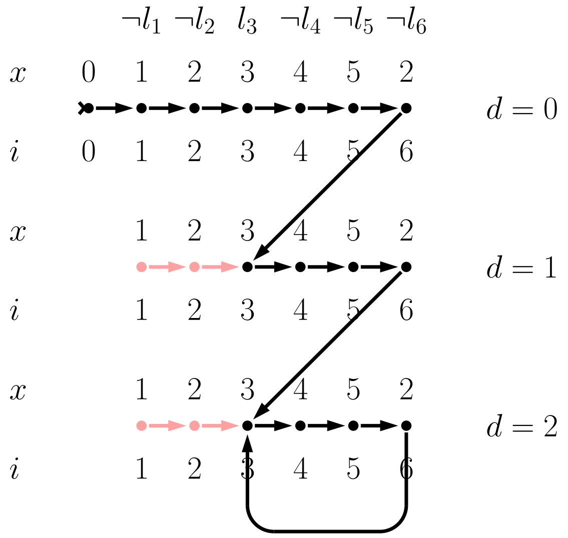

Figure 4.2: Black arcs show the Kripke structure induced by virtual unrolling of the loop for $k = 6$ up to depth 2 (i.e., $\delta(\psi) = 2$) when l_3 holds

$m - p(\pi) < n \leq i$. Thus, $\pi^i \models \psi$. The rest of the past temporal operators can be proven in a similar manner.

The specific part of proposition is a direct corollary of the first part. □

Benedetti and Cimatti [11] used this observation to design a BMC encoding for PLTL. The basic idea was to extend the standard BMC encoding for LTL by *virtually unrolling* the (k, l)-loop $\delta(\psi)$ times. Figure 4.2 depicts the situation for $\delta(\psi) = 2$ with the simple counter system. For the virtually unrolled (k, l)-loop it is again fairly straightforward to encode the semantics of the past operators. The future operators behave as before since the evaluations for the subformulas in the loop have stabilised in the $\delta(\psi)$-unrolling.

The encoding in [**P5**] also uses virtual unrolling for encoding PLTL. The past operators are encoded by using their recursive characterisations. Unrolling of the (k, l)-loop is accomplished by copying each subformula once for each unrolling of the loop. The loop selector variables make encoding the unrolled (k, l)-loop fairly straightforward. If we consider the $(6, 3)$-loop in Fig 4.2, past operators are evaluated along the path obtained by traversing the straight black arrows in the reverse direction.

The new encoding is very efficient and performs well. Experimental comparison [**P5**] with the original encoding of [11] shows that the new encoding clearly outperforms the original one. The size of the new encoding is $O(|I| + k|T| + k|\psi| \cdot \delta(\psi))$. The additional $\delta(\psi)$-term compared with the future case is due to virtual unrolling. Virtual unrolling facilitates detection of minimal counterexamples but it comes at a price. The new encoding is quadratic w.r.t. the size of the formula in the worst case. A BMC encoding for PLTL that does not perform virtual unrolling was presented in [30]. The encoding can miss minimal counterexamples but the size of the encoding is linear in $|\psi|$. The paper contained some errors in the presented encoding. The encoding would erroneously report a counterexample in the simple counter model of Figure 4.1 for the formula $\neg\mathbf{G}\,\mathbf{F}\,\mathbf{Y}\ (x = 2)$. However, the encoding can be fairly easily fixed [27].

PLTL has also been considered in other contexts than BMC. Lichtenstein et al. [120] described the first decision procedure for full PLTL. A symbolic tableau algorithm for constructing a Büchi automaton was presented in [109]. The paper also described symbolic algorithms for efficiently model checking systems with strong fairness constraints. In the BMC context fairness can be easily and efficiently dealt with by integrating the fairness constraints in the LTL specification. Schneider [141] has also considered the problem of generating a symbolic Büchi automaton for full PLTL. He shows several ways of producing optimised symbolic Büchi automata. The presented procedure is able to produce weak Büchi automata (see Section 3.3.2) from a large class of formulas. Schuppan and Biere [143] consider the problem of finding the shortest counterexamples when model checking PLTL. They analyse the capability of different symbolic Büchi automata constructions to accept minimal counterexamples. Additionally, using the liveness to safety construction [142] they present a BDD model checking procedure that accepts minimal counterexamples based on an adapted version of [**P5**].

5 CONCLUSIONS

The goal of this work has been developing efficient methods for model checking. We have focused on the algorithmic aspects of model checking. The two central themes of this work are the automata-theoretic approach to model checking and bounded model checking.

In publication [**P1**] we investigate the benefits of treating safety properties as a special case in LTL model checking. We think that treating safety as a special case is worth the effort. Our results show that deterministic automata can lead to smaller product state spaces than smaller non-deterministic automata expressing the same property. In addition, the first implementation for checking whether a formula is pathologic is presented.

Publication [**P2**] describes how modular analysis, a state space reduction method, can efficiently be combined with LTL-X model checking. The tester verification problem that results from the construction in [**P2**] is solved in full generality in publication [**P3**]. We analyse the tester verification problem and present a memory efficient algorithm that is proven correct.

Publications [**P4, P5**] consider efficient encodings for bounded model checking. In [**P4**] an encoding of the LTL bounded model checking problem linear in the size of the system, the size of the specification and the length of the bound is presented. Experimental results confirm that the encoding is more compact and efficient than previous encodings. In [**P5**] we generalise our encoding to bounded model checking PLTL. The generalised encoding is also linear in the length of the bound but quadratic in the size of the specification.

5.1 TOPICS FOR FURTHER RESEARCH

To further develop faster algorithms for verification with testers, another look at SCC-based algorithms is warranted. For solving the generalised Büchi emptiness problem, SCC-based algorithms seem to be very suitable [144]. They could possibly be extended to also solve the tester verification problem efficiently.

Model checking for modular Petri nets could be developed by refining the visibility concept. Since the method is fairly efficient at pruning states from the reachability graph, combining it with a method pruning arcs from the reachability graph could result in a significant improvement. Katz and Miller [105] show promising results for combining partial order methods with pruning invisible transitions.

Both the LTL and PLTL BMC encodings describe monotonic Boolean circuits when the state variables, their negations and the loop selector variables are seen as inputs of the circuit. This could be utilised in SAT solver optimisations.

Promising results for bounded model checking have been obtained using incremental SAT solvers [171, 154, 52, 101]. The basic idea is that since two BMC instances $|[M, \psi, k]|_k$ and $|[M, \psi, k]|_{k+1}$ are so similar, the solver could learn from the unsatisfiability proof of the earlier instances. The simplicity and clarity of our LTL and PLTL encodings make them amenable for an incremental encoding where many clauses could easily be forwarded to the next round.

One of the biggest shortcomings of bounded model checking is that it is an incomplete method. The method can find bugs, but cannot prove their absence. Completeness can be achieved by proving that after a bound k no counterexample can be found. One way of achieving this is to combine the inductive method of [52] with the safety to liveness transformation of [142]. Another possible approach would be to apply a Büchi automata based method similar to the method presented by Awedh and Somenzi [8]. In general, the possibility of exploiting optimisations developed for generating small Büchi automata [141, 64, 156] in BMC should be investigated further. After the submission of this work, we have developed an incremental encoding supporting induction for full PLTL [85].

BIBLIOGRAPHY

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on SAT-solvers. In Graf and Schwartzbach [72], pages 411–425.

[3] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[4] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[5] Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In Jensen and Podelski [99], pages 467–481.

[6] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The ForSpec temporal logic: A new temporal property-specification language. In Katoen and Stevens [104], pages 296–211.

[7] Roy Armoni, Limor Fix, Ranan Fraer, Scott Huddleston, Nir Piterman, and Moshe Y. Vardi. SAT-based induction for temporal safety properties. In Biere and Strichman [16].

[8] Mohammad Awedh and Fabio Somenzi. Proving more properties with bounded model checking. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 96–108. Springer, 2004.

[9] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In Berry et al. [13], pages 260–264.

[10] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The temporal logic Sugar. In Berry et al. [13], pages 363–367.

[11] Marco Benedetti and Alessandro Cimatti. Bounded model checking for past LTL. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2003.

[12] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer, 2001.

[13] Gérard Berry, Hubert Comon, and Alain Finkel, editors. *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*. Springer, 2001.

[14] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[15] Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu. Verifiying safety properties of a Power PC microprocessor using symbolic model checking without BDDs. In Halbwachs and Peled [79], pages 60–71.

[16] Armin Biere and Ofer Strichman, editors. *Bounded Model Checking, Second International Workshop, July 18, 2004, Boston, Massachusetts, USA*, 2004.

[17] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In Berry et al. [13], pages 454–464.

[18] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. In Hunt Jr. and Johnson [93], pages 37–54.

[19] Roderick Bloem, Kavita Ravi, and Fabio Somenzi. Efficient decision procedures for model checking of linear time logic properties. In Halbwachs and Peled [79], pages 222–235.

[20] Lubos Brim, Ivana Cerná, and Martin Necesal. Randomization helps in LTL model checking. In Luca de Alfaro and Stephen Gilmore, editors, *PAPM-PROBMIV*, volume 2165 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2001.

[21] Ed Brinksma and Kim Guldstrand Larsen, editors. *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*. Springer, 2002.

[22] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 38(8):677–691, 1986.

[23] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Symbolic model checking with partitioned transistion relations. In *VLSI*, volume A-1 of *IFIP Transactions*, pages 49–58. North-Holland, 1991.

[24] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[25] Ivana Cerná and Radek Pelánek. Relating hierarchy of temporal properties to model checking. In Branislav Rovan and Peter Vojtás, editors, *MFCS*, volume 2747 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 2003.

[26] Søren Christensen and Laure Petrucci. Modular analysis of Petri Nets. *The Computer Journal*, 43(3):224–242, 2000.

[27] Alessandro Cimatti. Personal communication, 2004.

[28] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Brinksma and Larsen [21], pages 359–364.

[29] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. Improving the encoding of LTL model checking into SAT. In Agostino Cortesi, editor, *VMCAI*, volume 2294 of *Lecture Notes in Computer Science*, pages 196–207. Springer, 2002.

[30] Alessandro Cimatti, Marco Roveri, and Daniel Sheridan. Bounded verification of past LTL. In Hu and Martin [92], pages 245–259.

[31] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization of skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[32] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[33] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[34] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[35] Edmund M. Clarke, Orna Grumberg, Kenneth L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *DAC*, pages 427–432. ACM Press, 1995.

[36] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[37] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In Steffen and Levi [153], pages 85–96.

[38] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362. IEEE Press, 1989.

[39] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In Berry et al. [13], pages 436–453.

[40] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448. ACM, 2000.

[41] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the Bandera specification language. *International Journal on Software Tools for Technology Transfer*, 4(1):34–56, 2002.

[42] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[43] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, 1996.

[44] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In Wing et al. [173], pages 253–271.

[45] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transaction on Programming Languages and Systems*, 19(2):253–291, 1997.

[46] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In Andrei Voronkov, editor, *CADE*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer, 2002.

[47] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification (extended abstract, category a). In Hunt Jr. and Somenzi [94], pages 14–26.

[48] Jörg Desel and Wolfgang Reisig. Place/Transition Petri nets. In Reisig and Rozenberg [136], pages 122–173.

[49] Matthew B. Dwyer, editor. *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings*, volume 2057 of *Lecture Notes in Computer Science*. Springer, 2001.

[50] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.

[51] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2–3):247–267, March 2004.

[52] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *First International Workshop on Bounded Model Checking*, volume 89 of *ENTCS*. Elsevier, 2003.

[53] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus. In *LICS* [96], pages 267–278.

[54] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[55] E. Allen Emerson and A. Prasad Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, 1984.

[56] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.

[57] Dawson R. Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In Steffen and Levi [153], pages 191–210.

[58] Javier Esparza. Decidability and complexity of Petri net problems - an introduction. In Reisig and Rozenberg [136], pages 374–428.

[59] Javier Esparza and Keijo Heljanko. Implementing LTL model checking with net unfoldings. In Dwyer [49], pages 37–56.

[60] Kousha Etessami and Gerard J. Holzmann. Optimizing Büchi automata. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2000.

[61] Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, and Zijiang Yang. Is there a best symbolic cycle-detection algorithm? In Margaria and Yi [124], pages 420–434.

[62] Alan M. Frisch, Daniel Sheridan, and Toby Walsh. A fixpoint based encoding for bounded model checking. In Mark Aagaard and John W. O'Leary, editors, *FMCAD*, volume 2517 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2002.

[63] Paul Gastin, Pierre Moro, and Marc Zeitoun. Minimization of counterexamples in SPIN. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2004.

[64] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Berry et al. [13], pages 53–65.

[65] Marc Geilen. On the construction of monitors for temporal logic properties. In *First Workshop on Runtime Verification*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[66] Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In Jensen and Podelski [99], pages 205–219.

[67] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *PSTV*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.

[68] Patrice Godefroid and Gerard J. Holzmann. On the verification of temporal properties. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *PSTV*, volume C-16 of *IFIP Transactions*, pages 109–124. North-Holland, 1993.

[69] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-space caching revisited. In Gregor von Bochmann and David K. Probst, editors, *CAV*, volume 663 of *Lecture Notes in Computer Science*, pages 178–191. Springer, 1992.

[70] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[71] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[72] Susanne Graf and Michael I. Schwartzbach, editors. *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Proceedings*, volume 1785 of *Lecture Notes in Computer Science*. Springer, 2000.

[73] Alex Groce and Daniel Kroening. Making the most of BMC counterexamples. In Biere and Strichman [16].

[74] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2003.

[75] Robert Groth. Is the software industry's productivity declining? *IEEE Software*, 21(65):92–94, 2004.

[76] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[77] Orna Grumberg, Assaf Schuster, and Avi Yadgar. Memory efficient all-solutions SAT solver and its application for reachability analysis. In Hu and Martin [92], pages 275–289.

[78] Viktor Gyuris and A. Prasad Sistla. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design*, 15(3):217–238, November 1999.

[79] Nicolas Halbwachs and Doron Peled, editors. *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*. Springer, 1999.

[80] Nicolas Halbwachs and Lenore D. Zuck, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*. Springer, 2005.

[81] Henri Hansen, Wojciech Penczek, and Antti Valmari. Stuttering-insensitive automata for on-the-fly detection livelock properties. In *Proceedings of the 7th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS 2002)*, volume 66(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[82] Peter E. Hart, Nils J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on on Systems Science and Cybernetics*, 4:100–107, 1968.

[83] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In Katoen and Stevens [104], pages 342–356.

[84] Keijo Heljanko. *Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets*. PhD thesis, Helsinki University of Technology, Department of Computer Science and Engineering, February 2002.

[85] Keijo Heljanko, Tommi Junttila, and Timo Latvala. Incremental and complete bounded model checking for full PLTL. In K. Etessami and S. Rajamani, editors, *Computer Aided Verification*, 2005. to appear.

[86] Juhana Helovuo and Antti Valmari. Checking for CFFD-preorder with tester processes. In Graf and Schwartzbach [72], pages 283–298.

[87] Monika Rauch Henzinger and Jan Arne Telle. Faster algorithms for the nonemptiness of streett automata and for communication protocol pruning. In Rolf G. Karlsson and Andrzej Lingas, editors, *SWAT*, volume 1097 of *Lecture Notes in Computer Science*, pages 16–27. Springer, 1996.

[88] Gerard. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[89] Gerard J. Holzmann. The logic of bugs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 81–87. ACM, 2002.

[90] Gerard J. Holzmann, Doron Peled, and Mihalis Yannakakis. On nested depth first search. In *Proceedings of the 2nd* SPIN *Workshop*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 23–32. American Mathematical Society, 1997.

[91] Gerard J. Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, 2002.

[92] Alan J. Hu and Andrew K. Martin, editors. *Formal Methods in Computer-Aided Design, 5th International Confrence, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, volume 3312 of *Lecture Notes in Computer Science*. Springer, 2004.

[93] Warren A. Hunt Jr. and Steven D. Johnson, editors. *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*. Springer, 2000.

[94] Warren A. Hunt Jr. and Fabio Somenzi, editors. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003.

[95] Nisse Husberg and Tapio Manner. Emma: Developing an industrial reachability analyser for SDL. In Wing et al. [173], pages 642–661.

[96] IEEE Computer Society. *Proceedings, Symposium on Logic in Computer Science, 16-18 June 1986, Cambridge, Massachusetts, USA*, 1986.

[97] IEEE Computer Society. *5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada*, 2001.

[98] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–76, 1996.

[99] Kurt Jensen and Andreas Podelski, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*. Springer, 2004.

[100] HoonSang Jin, Kavita Ravi, and Fabio Somenzi. Fate and free will in error traces. In Katoen and Stevens [104], pages 445–459.

[101] HoonSang Jin and Fabio Somenzi. An incremental algorithm to check satisfiability for bounded model checking. In Biere and Strichman [16].

[102] Tommi Junttila. On the symmetry reduction method for Petri nets and similar formalisms. Research Report A80, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, September 2003. Doctoral dissertation.

[103] Roope Kaivola. Equivalences, preorders and compositional verification for linear time temporal logic and concurrent systems. Report A-1996-1, University of Helsinki, Department of Computer Science, 1996. PhD thesis.

[104] Joost-Pieter Katoen and Perdita Stevens, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*. Springer, 2002.

[105] Shmuel Katz and Hillel Miller. Saving space by fully exploiting invisible transitions. *Formal Methods in System Design*, 14(3):311–332, 1999.

[106] Shmuel Katz and Doron Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107–120, 1992.

[107] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363. John Wiley and Sons, 1992.

[108] Yonit Kesten and Amir Pnueli. Verification by augmented finitary abstraction. *Inf. Comput.*, 163(1):203–243, 2000.

[109] Yonit Kesten, Amir Pnueli, and Li-on Raviv. Algorithmic verification of linear temporal logic specifications. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[110] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *VMCAI*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003.

[111] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Pushdown specifications. In Matthias Baaz and Andrei Voronkov, editors, *LPAR*, volume 2514 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2002.

[112] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[113] Orna Kupferman and Moshe Y. Vardi. From complementation to certification. In Jensen and Podelski [99], pages 591–606.

[114] Robert P. Kurshan. Complementing deterministic Büchi automata in polynomial time. *Journal of Computer and System Science*, 35(1):59–71, 1987.

[115] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[116] Robert P. Kurshan, Vladimir Levin, and Hüsnü Yenigün. Compressing transitions for model checking. In Brinksma and Larsen [21], pages 569–581.

[117] Leslie Lamport. What good is temporal logic. In R.E.A. Mason, editor, *IFIP 9th World Congress*, pages 657–668. North-Holland, 1983.

[118] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Temporal logic with forgettable past. In *LICS*, pages 383–392. IEEE Computer Society, 2002.

[119] Timo Latvala and Keijo Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1-4):175–193, 2000.

[120] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[121] Jørn Lind-Nielsen, Henrik Reif Andersen, Gerd Behrmann, Henrik Hulgaard, Kåre J. Kristoffersen, and Kim Guldstrand Larsen. Verification of large state/event systems using compositionality and dependency analysis. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 1998.

[122] Marko Mäkelä. Maria: Modular reachability analyser for algebraic system nets. In Javier Esparza and Charles Lakos, editors, *ICATPN*, volume 2360 of *Lecture Notes in Computer Science*, pages 434–444. Springer, 2002.

[123] Marko Mäkelä. Model checking safety properties in modular high-level nets. In Wil M. P. van der Aalst and Eike Best, editors, *ICATPN*, volume 2679 of *Lecture Notes in Computer Science*, pages 201–220. Springer, 2003.

[124] Tiziana Margaria and Wang Yi, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001, Proceedings*, volume 2031 of *Lecture Notes in Computer Science*. Springer, 2001.

[125] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In Gregor von Bochmann and David K. Probst, editors, *CAV*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 1992.

[126] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[127] Kenneth L. McMillan. Interpolation and SAT-based model checking. In Hunt Jr. and Somenzi [94], pages 1–13.

[128] Kedar S. Namjoshi. Certifying model checkers. In Berry et al. [13], pages 2–13.

[129] Amit Narayan, Adrian J. Isles, Jawahar Jain, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Reachability analysis using partitioned-ROBDDs. In *ICCAD*, pages 388–393. ACM and IEEE Computer Society, 1997.

[130] Christos Papadimitriou. *Computational Complexity*. Addision-Wesley, 1994.

[131] Doron Peled and Lenore D. Zuck. From model checking to a temporal proof. In Dwyer [49], pages 1–14.

[132] Wojciech Penczek, Bozena Wozna, and Andrzej Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundam. Inform.*, 51(1-2):135–156, 2002.

[133] Amir Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[134] Jean-Pierre Quielle and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pages 337–350, 1981.

[135] Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In Jensen and Podelski [99], pages 31–45.

[136] Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998.

[137] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on computer-aided design*, pages 42–47. IEEE Computer Society Press, 1993.

[138] Theo C. Ruys and Ed Brinksma. Managing the verification trajectory. *International Journal on Software Tools for Technology Transfer*, 4(1):246–259, 2003.

[139] Shmuel Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institute of Science, 1989.

[140] Karsten Schmidt. How to calculate symmetries of Petri nets. *Acta Informatica*, 36(7):545–590, 2000.

[141] Klaus Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2001.

[142] Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer*, 5(2–3):185–204, March 2004.

[143] Viktor Schuppan and Armin Biere. Shortest counterexamples for symbolic model checking of LTL with past. In Halbwachs and Zuck [80], pages 493–509.

[144] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In Halbwachs and Zuck [80], pages 174–190.

[145] Roberto Sebastiani and Stefano Tonetta. "more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In Daniel Geist and Enrico Tronci, editors, *CHARME*, volume 2860 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2003.

[146] Koushik Sen, Grigore Rosu, and Gul Agha. Generating optimal linear temporal logic monitors by coinduction. In Vijay A. Saraswat, editor, *ASIAN*, volume 2896 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2003.

[147] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Hunt Jr. and Johnson [93], pages 108–125.

[148] ShengYu Shen, Ying Qin, and Sikun Li. Minimizing counterexample with unit core extraction and incremental SAT. In Radhia Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2005.

[149] Daniel Sheridan. Bounded model checking with SNF, alternating automata and Büchi automata. In Biere and Strichman [16].

[150] A. Prasad Sistla. Safety, liveness, and fairness in temporal logic. *Formal Aspects in Computing*, 6:495–511, 1994.

[151] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[152] Margaret H. Smith, Gerard J. Holzmann, and Kousha Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *RE* [97], pages 14–22.

[153] Bernhard Steffen and Giorgio Levi, editors. *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*. Springer, 2004.

[154] Ofer Strichman. Pruning techniques for the SAT-based bounded model checking problem. In Tiziana Margaria and Thomas F. Melham, editors, *CHARME*, volume 2144 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2001.

[155] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[156] Heikki Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, 2003.

[157] Heikki Tauriainen. Nested emptiness search for generalized Büchi automata. In *ACSD*, pages 165–174. IEEE Computer Society, 2004.

[158] Heikki Tauriainen and Keijo Heljanko. Testing LTL formula translation into Büchi automata. *STTT - International Journal on Software Tools for Technology Transfer*, 4(1):57–70, 2002.

[159] Wolfgang Thomas. A combinatorial approach to the theory of ω-automata. *Information and Control*, 48(3):261–283, 1981.

[160] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science: Formal Models and Semantics*, volume B, pages 133–191. Elsevier, 1990.

[161] Teemu Tynjälä, Sari Leppänen, and Vesa Luukkala. Verifying reliable data transmission over UMTS radio interface with high level Petri nets. In Doron Peled and Moshe Y. Vardi, editors, *FORTE*, volume 2529 of *Lecture Notes in Computer Science*, pages 178–193. Springer, 2002.

[162] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[163] Antti Valmari. On-the-fly verification with stubborn sets. In Costas Courcoubetis, editor, *CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 397–408. Springer, 1993.

[164] Antti Valmari. The state explosion problem. In Reisig and Rozenberg [136], pages 429–528.

[165] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE* [97], pages 249–262.

[166] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.

[167] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Margaria and Yi [124], pages 1–22.

[168] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS* [96], pages 332–344.

[169] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[170] Kimmo Varpaaniemi. On stubborn sets in the verification of linear time temporal properties. *Formal Methods in System Design*, 26(1):45–67, January 2005.

[171] Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. SATIRE: A new incremental satisfiability engine. In *DAC*, pages 542–545. ACM, 2001.

[172] Poul F. Williams. *Formal Verification Based on Boolean Expression Diagrams*. PhD thesis, Dept. of Information Technology, Technical University of Denmark, Lyngby, Denmark, 2000.

[173] Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors. *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I*, volume 1708 of *Lecture Notes in Computer Science*. Springer, 1999.

[174] Bozena Wozna. ACTLS properties and bounded model checking. *Fundam. Inform.*, 63(1):65–87, 2004.

[175] Karen Yorav. *Exploiting Syntactic Structure for Automatic Verification*. PhD thesis, Israel Institute of Technology, 2000.

A ADDITIONS AND CORRECTIONS TO PUBLICATIONS

Publication [**P1**]:

- On page 77 the claim that “fairness need not be taken into account, if we know we are dealing with a safety specification.” is only true for *machine closed* [1] fairness specifications. A fairness property is machine closed if does not rule out any finite behaviour. For instance, fair Kripke structures are not machine closed.

- The pseudocode algorithm description starting on page 79 has two errors: the removal of σ from S' should be done after the **od** on the next line, and we should not add elements to X on the last but third line without checking the if the element has been seen before (the set Q). See Figure 3.1 on page 17 for the complete and corrected pseudocode.

Publication [**P2**]:

- The Petri net in Figure 1 on page 300 has a bug: the arc from the place “busy” to the transition “go_crit_r” should be reversed

- In the proof of Theorem 1 on page 307, part (a) contains a numbering inconsistency. The sentence “otherwise if $t_i \in T_V$ set $j = j + 1$” should read “otherwise (iii) if $t_i \in T_V$ set $j = j + 1$”. This makes the numbering consistent with the description of the steps that follows.

- On page 307, the claim that “… we then have a solution which traverses the product three times in the worst case by first using Valmari’s one-pass algorithm any illegal livelocks. If no illegal livelocks are found we can use the standard nested depth first algorithm to find any illegal infinite traces.” is incorrect. The best known algorithm for the tester verification requires four passes of the state space [**P3**].

Publication [**P4**]:

- In Section 3, the table defining the translation is missing cases for the $\wedge$-operator and the $\vee$-operator. The missing cases are:

$:=$	$i \leq k$	$i = k + 1$
$\lvert[\psi_1 \wedge \psi_2]\rvert_i$	$\lvert[\psi_1]\rvert_i \wedge \lvert[\psi_2]\rvert_i$	$\lvert[\psi_1]\rvert_i \wedge \lvert[\psi_2]\rvert_i$
$\lvert[\psi_1 \vee \psi_2]\rvert_i$	$\lvert[\psi_1]\rvert_i \vee \lvert[\psi_2]\rvert_i$	$\lvert[\psi_1]\rvert_i \vee \lvert[\psi_2]\rvert_i$

- In Section 3.1, in the table defining the basic optimised encoding, all occurrences of φ should be replaced by ψ. For completeness the correct table is reproduced below.

$:=$	$i \leq k$	$i = k + 1$
$\lvert[\mathbf{G}\,\psi]\rvert_i$	$\lvert[\psi]\rvert_i \wedge \lvert[\mathbf{G}\,\psi]\rvert_{i+1}$	$\bigvee_{j=1}^{k} \left(l_j \wedge \langle\langle \mathbf{G}\,\psi \rangle\rangle_j \right)$
$\lvert[\mathbf{F}\,\psi]\rvert_i$	$\lvert[\psi]\rvert_i \vee \lvert[\mathbf{F}\,\psi]\rvert_{i+1}$	$\bigvee_{j=1}^{k} \left(l_j \wedge \langle\langle \mathbf{F}\,\psi \rangle\rangle_j \right)$
$\lvert[\mathbf{G}\,\mathbf{F}\,\psi]\rvert_i$	$\lvert[\mathbf{G}\,\mathbf{F}\,\psi]\rvert_{k+1}$	$\bigvee_{j=1}^{k} \left(l_j \wedge \langle\langle \mathbf{F}\,\psi \rangle\rangle_j \right)$
$\lvert[\mathbf{F}\,\mathbf{G}\,\psi]\rvert_i$	$\lvert[\mathbf{F}\,\mathbf{G}\,\psi]\rvert_{k+1}$	$\bigvee_{j=1}^{k} \left(l_j \wedge \langle\langle \mathbf{G}\,\psi \rangle\rangle_j \right)$
$\langle\langle \mathbf{G}\,\psi \rangle\rangle_i$	$\lvert[\psi]\rvert_i \wedge \langle\langle \mathbf{G}\,\psi \rangle\rangle_{i+1}$	$\top$
$\langle\langle \mathbf{F}\,\psi \rangle\rangle_i$	$\lvert[\psi]\rvert_i \vee \langle\langle \mathbf{F}\,\psi \rangle\rangle_{i+1}$	$\bot$

Publication [**P5**]:

- On page 382, Section 2.1, the definition of π^i is not compatible with the definition of PLTL in Section 3.1. This can be remedied by replacing the sentence "Further, let π^i denote the suffix of π starting from the i:th state." with "Further, let $\pi^i \models \psi$ denote the validity of ψ when evaluated at position i of π.".

- In Section 3.2, page 388, in the table at the top of the page the $\delta(\psi)$ should be $\delta(\varphi)$. Thus, the second column of the first row should read $0 \leq d \leq \delta(\varphi), 0 \leq i \leq k$.

- In Section 5 on page 395, we claim that "We not see how soundness for full PLTL could be achieved without performing virtual unrolling.". It is actually possible. The encoding would need to be modified so that a loop could only be closed if $s_k = s_l$ *and* all subformulas would have the same truth values at k and l. The exact details are left for further work.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE RESEARCH REPORTS

HUT-TCS-A82 Tomi Janhunen
Translatability and Intranslatability Results for Certain Classes of Logic Programs.
November 2003.

HUT-TCS-A83 Heikki Tauriainen
On Translating Linear Temporal Logic into Alternating and Nondeterministic Automata.
December 2003.

HUT-TCS-A84 Johan Wallén
On the Differential and Linear Properties of Addition. December 2003.

HUT-TCS-A85 Emilia Oikarinen
Testing the Equivalence of Disjunctive Logic Programs. December 2003.

HUT-TCS-A86 Tommi Syrjänen
Logic Programming with Cardinality Constraints. December 2003.

HUT-TCS-A87 Harri Haanpää, Patric R. J. Östergård
Sets in Abelian Groups with Distinct Sums of Pairs. February 2004.

HUT-TCS-A88 Harri Haanpää
Minimum Sum and Difference Covers of Abelian Groups. February 2004.

HUT-TCS-A89 Harri Haanpää
Constructing Certain Combinatorial Structures by Computational Methods. February 2004.

HUT-TCS-A90 Matti Järvisalo
Proof Complexity of Cut-Based Tableaux for Boolean Circuit Satisfiability Checking.
March 2004.

HUT-TCS-A91 Mikko Särelä
Measuring the Effects of Mobility on Reactive Ad Hoc Routing Protocols. May 2004.

HUT-TCS-A92 Timo Latvala, Armin Biere, Keijo Heljanko, Tommi Junttila
Simple Bounded LTL Model Checking. July 2004.

HUT-TCS-A93 Tuomo Pyhälä
Specification-Based Test Selection in Formal Conformance Testing. August 2004.

HUT-TCS-A94 Petteri Kaski
Algorithms for Classification of Combinatorial Objects. June 2005.

ISBN 951-22-7787-5
ISBN 951-22-7788-3 (PDF)
ISSN 1457-7615